TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

Matti Riihimäki

Swift-ohjelmointikielen käyttäminen iOS-sovelluskehityksessä

Diplomityö

Tarkastaja: Tommi Mikkonen
Tarkastaja ja aihe hyväksytty
09.11.2016

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Master's Degree Programme in Pervasive computing
RIIHIMÄKI, MATTI: Using Swift in iOS application development
Master of Science Thesis, 49 pages
December 2016
Major: Software Development
Examiner: prof. Tommi Mikkonen
Keywords: Swift, iOS, Apple, Software, Programmin language

Developing software for iOS -platform has changed in what comes to programming language. Apple introduced a new programming language called Swift in summer 2014. Before that the only programming language that could be used for developing software for iOS was Objective-C. The purpose of this master thesis is to find out if the new language can be used for commercial programs. The thesis also some what compares the new language to the old one. The main point is answering to the question "Should Swift be used to develop iOS-software".

This master thesis will tell what you should consider while making the decision. From a project point of view people play a vital role in this. Thesis goes through the three options that there is: to use only one of the languages or both. Swift language is presented by comparing some features to Objective-C. The main idea is not to compare these two languages rather than to show what's new in Swift and if Objective-C has the same functionality to illustrate what's different. Result of answering the question has two parts. One is the interviews of three iOS-developers that have different kind of projects. To support the interviews, thesis shows findings from the internet. What the people online have to say about the Swift language and what they think about Objective-C? At the end of the thesis there is a summary where the presented questions will have answers.

Even though Swift is only a two and half year old language it is ready and should be used. Objective-C is about thirty years old and time has taken a toll (even though it's not the only reason) and Objective-C has not been able to keep up with the progress. One big reason that came up in all the interviews and in the internet was the syntax. Everybody liked Swifts new and light syntax compared to clumsy and long Objective-C syntax. There are many reasons and they are presented in this thesis, but to sum it up, Swift is the language that should be used for new and old projects in iOS-development. Swift is the future and Objective-C is the past.

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO
Tietotekniikan koulutusohjelma
RIIHIMÄKI, MATTI: Swift-ohjelmointikielen käyttäminen iOS-sovelluskehityksessä
Diplomityö, 49 sivua
Joulukuu 2016
Pääaine: Ohjelmistotuotanto
Tarkastaja: prof. Tommi Mikkonen
Avainsanat: Swift, iOS, Apple, Ohjelmointikieli, Ohjelmistotuotanto

iOS-sovelluskehittäminen on suuren muutoksen keskellä. Enää ei ole pakko käyttää Objective-C-kieltä. Apple on julkisti kesällä 2014 ohjelmointikielen nimeltään Swift. Swift on aivan uusi ohjelmointikieli ja tällä hetkellä kaksi ja puoli vuotta vanha. Työssä on tarkoitus selvittää, onko Swift-ohjelmointikieli tarpeeksi kypsä, jotta sitä voitaisiin käyttää iOS-sovelluskehittämisessä. Onko se ylipäätäänsä kieli, mitä halutaan käyttää sovellusten kehittämiseen, vai onko Objective-C-kieli vielä se ohjelmointikieli, millä sovellukset kannattaa kehittää?

Täsä diplomityössä esitellään kumpaa ohjelmointikieltä tulee käytää missäkin tilanteessa. Työssä tarkastellaan asiaa myös niin projektin kuin henkilöstän kannalta. Projetin kannalta niin, että asia kuten aika, raha, osaaminen tulee huomioitua. Henkilöstön tapauskessa asiaa tarkestellaan kuinka halukkaita vaihtaa kieltä ja minkälainen tausta heillä on liittyen ohjelmointikieliin. Työssä myös käydään läpi kaikki kolme vaihtoehtoa eli käytetään pelkästään toista kieltä tai käytetään molempia kieliä. Työssä esitellään Swift-ohjelmointikielen uudet ominaisuudet ja verrataan ominaisuuksia Objective-C-kieleen jos niitä on siinä olemassa. Tulokset ja väittämät ovat saatu haastatteluiden ja verkkosivuen kautta.

Swift-ohjelmointikieli on tämän työn kirjoitus hetkellä kaksi ja puoli-vuotias. Tämän vuoksi työssä pyritään vastaamaan kysymykseen voiko Swift-ohjelmointikieltä käyttää iOS-sovelluksien kehittämiseen. Asiaa tarkastellaan sen kannalta, onko ohjelmointikieli tarpeeksi kypsä käytettäväksi kaupallisissa ohjelmistoprojekteissa. Onko Swift parempi ohjelmointikieli kuin Objective-C-kieli, joka on ennen ollut ainut vaihtoehto iOS-sovelluskehittämiseen? Työn lopputulema on, että Swift-ohjelmointikieli on tulevaisuus, ja sitä tulisi käyttää. Objective-C-kieli on tällä hetkellä käytössä monissa projekteissa, ja projektista riippuen saattaa sen käytön jatkaminen olla perusteltua. Työssä kuitenkin ehdotetaan, että uusissa projekteissa tulisi käyttää Swift-ohjelmointikieltä.

ALKUSANAT

Aloitin diplomityöni tekemisen tammikuussa 2015. Minulle kuitenkin kävi kuten monille opiskelijoille, eli työ ja elämä veivät huomioni. Tammikuussa 2016 havahduin siihen, että olin viimeksi työstänyt diplomityötäni puoli vuotta sitten. Saman vuoden kesällä päätin ottaa kahden viikon diplomityöloman, jotta saisin puuttuvat sivut kasaan. Sain sivut kasaan, mutta työ ei ollut sellainen kuin halusin. Diplomityönohjaajaani Tommi Mikkonen oli samaa mieltä. Mikkosen antaman palautteen avulla sain uuden suunnan diplomityölleni ja aloitin työn alusta uudestaan. Nyt syksyllä 2016 valmistunutta diplomityötä lukiessani olen todella tyytyväinen päätökseeni.

Kymmenen vuoden opiskelu tulee tämän työn myötä päätökseen. Haluankin kiittää vanhempiani Eine Rosenberg-Riihimäkeä ja Juhani Riihimäkeä tuesta, jota he ovat antaneet minulle opiskeluideni aikana. Ilman heitä tämä ei olisi ollut mahdollista.

Lopputyöni kannalta haluan kiittää ohjaajani Tommi Mikkosta, jonka rakentava palautteen avulla sain työstä sellaisen, kuin olin siitä halunnut. Haluan myös kiittää työystäviäni Lari Kolia, Matti Määttää ja Saija Saarenpäätä suostumisesta haastatteluihin.

Lopuksi haluan kiittää kaikkia joiden kanssa olen ollut tekemisissä opiskeluaikanani. Ilman teitä en olisi viihtynyt täällä näin kauan.

Tampere 20.11.2016 Matti Riihimäki

SISÄLTÖ

1. JOHDANTO

iOS-sovelluskehittämisessä on ollut yksi vaihtoehto ohjelmointikieleksi. Objective-C-kieli on aiemmin ollut ainut vaihtoehto, mutta tämä muuttui, kun Apple julkisti Swift-ohjelmointikielen kesällä 2014[29]. Koska Swift-ohjelmointikieli on niin uusi, herää kysymys siitä, voiko sitä käyttää kaupallisessa ohjelmistoprojekteissa. Mitä vanhoille sovelluksille käy, ja kuinka Objective-C-kieli toimii Swift-ohjelmointikielen kanssa?

Diplomityössä on kuvataan kumpaa ohjelmointikieltä tulee käyttää ja missä tilanteissa. Työssä on myös esitelty Swift-ohjelmointikielen käyttöön liittyviä riskejä ja mahdollisia ongelmia.

Luvussa 2 käydään läpi tausta ja nykytilanne liittyen iOS-sovelluskehitykseen. Luvussa käydään läpi mitä asioita liittyy Applen iOS-käyttöjärjestelmälle kehittämiseen ja käyttöjärjestelmälle kehittämisen nykytilannetta.

Luvussa 3 esitellään minkälainen Swift-ohjelmointikieli on. Luvussa on esitelty kielen tärkeimpiä ominaisuuksia ja erityispiirteitä. Luku on kirjoitettu siinä mielessä, että Swift tulee olemaan vaihtoehto Objective-C-kielelle. Tämän takia on pyritty löytämään Swift-ohjelmointikielen tärkeimmät ominaisuudet, ja ne ominaisuudet mitkä eroavat Objective-C-kielestä ilman, että näitä kahta suoranaisesti vertailtaisiin keskenään.

Luvussa 4 on selvitetty mitä mieltä verkossa ollaan Swift-ohjelmointikielestä. Luvussa on kerätty eri lähteistä tietoa siitä, miten Swift-ohjelmointikieli on otettu vastaan käyttäjien parissa, ja mitä siitä on oltu mieltä.

Luvussa 5 on haastateltu kolmea kehittäjää. Kehittäjät kertovat kokemuksistaan liittyen iOS-sovelluskehittämiseen. Haastatteluja on kolme kappaletta ja jokaisessa niistä on oma näkökulmansa. Näkökulmat ovat pelkän Objective-C-kielen käyttäminen, pelkän Swift-ohjelmointikielen käyttäminen ja viimeisenä molempien kielien käyttäminen samassa projektissa. Haastatteluissa on pyritty löytämään oleelliset asiat kuhunkin näkökulmaan liittyen.

Luku 6, joka on työn viimeinen luku, sisältää yhteenvedon ja vastauksen kysymykseen voiko Swift-ohjelmointikieltä käyttää kaupallisessa projektissa.

2. TAUSTAA

Mikäli haluaa kehittää sovelluksia iPhonelle, iPadille tai muille Applen tuotteille, on käytettävä Objective-C-ohjelmointikieltä. Tähän tuli muutos kesällä 2014, kun Apple julkisti uuden ohjelmointikielen, Swiftin. Applen tuotteille kehittämisessä on muitakin Applen määräämiä asoita kuin ohjelmointikieli. Lisenssi on maksullinen, ja sitä tulee hakea erikseen. Sovellukset levitetään AppStoren kautta. Kehittäjän tulee siis lisätä sovellus AppleStoreen, jotta käyttäjät voivat ladata sovelluksen. Nykyään on myös mahdollista ns. talon sisäiset kaupat eli halutessaan voidaan itse pystyttää palvelin, johon vain rajattu joukko (yrityksen työntekijät) pääsee. Tämä on tietysti mahdollista vain erikoisluvalla, joka tulee hakea erikseen.[35]

2.1 Apple

Steve Jobs, Steve Wozniak ja Ronald Wayne perustivat Apple Computer -yrityksen 1.4.1976 [14]. Nykyään yrityksen nimi on vain Apple, sillä yritys tekee muutakin kuin valmistaa tietokoneita. Apple on nykyisin monikansallinen yritys, joka valmistaa tietokoneiden lisäksi mm. mobiililaitteita kuten iPhone ja iPad. Laitteiden lisäksi yritys myy digitaalista mediaa kuten elokuvia ja musiikkia. Yritys on siis muuttunut ja kehittynyt paljon siitä, mitä varten se alun perin perustettiin.

Applen valmistamissa tietokoneissa on käytössä Applen oma käyttöjärjestelmä macOS, missä on UNIX ydin[12]. Käyttöjärjestelmä on Applen itse kehittämä, ja siitä ilmestyy vuosittain uusi versio, joka on viime vuosina ollut ilmainen. Yrityksellä on myös oma mobiilikäyttöjärjestelmä iOS. iOS-käyttöjärjestelmä on käytössä yrityksen valmistamissa iPhone- ja iPad-laitteissa. Näiden lisäksi uusia tulokkaita ovat tvOS[30] sekä watchOS[31]. Kuten nimi vihjaa, on ensiksi mainittu käytössä Applen valmistamalla televisioon kytkettävällä lisälaitteella ja jälkimmäinen Applen valmistamissa rannekelloissa.

2.2 iOS-mobiilikäyttöjärjestelmä

Vuonna 2007 ensimmäinen iPhone julkistettiin ja samalla esiteltiin iOS-mobiilikäyttöjärjestelmä[5]. iOS on Applen itse suunnittelema, ja näin ollen natiivikieli sille kehittämiseen on ollut kesään 2014 saakka Objective-C. Kesällä 2014 Apple julkisti uuden ohjelmointikielen nimeltä Swift[29]. Swift on siis nykyään vaihtoehto natiivien sovellusten kirjoittamiseen Applen tuotteille. Näin ollen myös iOS-

mobiilikäyttöjärjestelmälle voidaan kirjoittaa sovelluksia Swift-ohjelmointikielen avulla. iOS on tarkoitettu käytettäväksi ainoastaan Applen laitteissa ja on ainut käyttöjärjestelmävaihtoehto Apple-mobiililaitteille[21].

2.3 Lisenssi

Mikäli haluaa kehittää sovelluksia iOS-alustalle, tulee kehittäjän hakea ja ostaa lisenssi Applelta. Lisenssejä on olemassa kolmea eri tyyppiä: yksityishenkilöille, yhtiöille ja yliopistoille.

Lisenssi[1] yksityiselle henkilölle maksaa $ 99, ja se on voimassa vuoden. Lisenssi oikeuttaa kehittäjän saamaan beta-vaiheessa olevia versioita iOS-mobiilikäyttöjärjestelmästä, sekä XCode-kehitysympäristöstä. Näiden lisäksi sovelluksen asentaminen fyysiselle laitteelle tulee mahdolliseksi, ja kehittäjä voi lähettää sovelluksen arvioitavaksi. Hyväksymisen jälkeen sovellus voidaan laittaa AppStoreen kaikkien käyttäjien saataville. Tässä kohtaa on hyvä huomata, että mikäli lisenssi päättyy, poistetaan jo AppStoreen laitettu sovellus sieltä.[23] Sovellusta voidaan jakaa käyttäjille vain AppStoren kautta, joten kuka tahansa voi sen sieltä käydä hakemassa.[24]

Yrityksille tarkoitettu lisenssi maksaa $299 vuodessa. Jotta lisenssiä voi hakea, tulee yrityksellä olla D-U-N-S numero, mikä vastaa Suomen Y-tunnusta. Erona yksityisille tarkoitettuun lisenssiin on, että sovelluksia voidaan jakaa yrityksen sisällä. Yritys voi siis luoda oman ”AppStoren” mihin pääsevät vain luvan saaneet.[22]

Yliopistoille tarkoitettu lisenssi on ilmainen. Tätä lisenssiä voi hakea vain yliopiston henkilökunta. Sen avulla voidaan saada maksimissaan 200 opiskelijan tiimejä professorin kanssa. Sovelluksia voidaan myös jakaa sähköpostien tai verkkosivujen kautta. Tätä tulee käyttää vain opetustarkoitukseessa, eli lähinnä arvosteluun liittyvissä tilanteissa.[25]

2.4 XCode

Xcode on Applen oma IDE, jota käytetään iOS ja OS X sovelluksien kehittämiseen. IDE on ilmainen, mutta jos on hankkinut lisenssin, on mahdollista ladata beta-versioita. Tärkeimpiä ominaisuuksia IDEssä ovat koodieditori, kääntäjä, käyttöliittymäsuunnittelutyökalu[2] ja dokumentaatio Applen kirjastoista kuten Cocoa Touch, jota käytetään iOS-alustan yhteydessä ja simulaattorina iOS-alustaa varten.[16]

Kääntäjä on Open-Source LLVM[26]. Kääntäjä tukee seuraavia kieliä: C, C++, Objective-C sekä Swift[16; 15]. Kääntäjää voidaan käyttää suoraan XCoden tai Terminal-sovelluksen kautta. Kääntäjä on optimoitu siten, että se tuottaa mahdollisimman nopeita ja tehokkaita sovelluksia Applen omille tuotteille.

[1] 9.5.2016

[2] Interface Builder

Käyttöliittymän suunnittelu onnistuu myös visuaalisesti XCode Interface Builder -ominaisuuden avulla. Käyttöliittymän komponentteja voidaan raahata paikoilleen, ja samalla nähdä visuaalisesti miten ne asettuvat toisiinsa nähden. Apuna komponenttien sijoitteluun on käytettävissä Auto Layout -toiminto. Auto Layoutin avulla voidaan määritellä rajoituksia, joiden avulla käyttöliittymän komponentit ovat eri kokoisilla näytöillä oikeissa kohdissa.

iOS-sovelluksia voidaan testata simulaattorin avulla. Tällöin sovellusta ei tarvitse joka kerta asentaa fyysiselle laitteelle. Tämä on myös ainut tapa testata sovellusta ennen kuin on hankkinut iOS-lisenssin Applelta.[16]

2.5 AppStore

Sovellukset tulevat kaikkien saataville AppStoreen, joka on Applen hallinnoima ja ylläpitämä järjestelmä. Sovelluksen pitäminen AppStoressa ei maksa mitään sinänsä, mutta mikäli kehittämistä varten tarvittava lisenssi menee vanhaksi, poistetaan jo AppStoressa olevat sovellukset ja myöskään uusia sovelluksia ei pysty sinne enää lisäämään. Näin ollen voidaan ajatella, että lisenssin hinta sisältää myös kustannukset siitä, että sovelluksen saa levitettyä käyttäjille AppStoren kautta.

Sovellukset jaetaan neljää eri ryhmään riippuen siitä, minkälaista sisältöä ne tarjoavat. Kriteereitä ovat mm. sisältääkö sovellus väkivaltaa, seksuaalista sisältöä, alkoholia, huumeita, tupakkaa, uhkapeliä tai kauhua. Sovelluksen kehittäjän täytettyä kyselyn AppStore kertoo mihin ikäsuositusryhmään sovellus tullaan sijoittamaan[13]. Ryhmät ovat jaettu iän mukaan ja ne ovat 4+, 9+, 12+ ja 17+.

AppStore on Applen ylläpitämä ja tarkistaa mitä sovelluksia se hyväksyy. AppStore pidättää oikeuden poistaa sovelluksen, mikäli se näkee sen tarpeelliseksi. Eri maiden välillä saattaa olla myös jonkin verran eroja sovelluksien saatavuudessa.

2.6 Objective-C

Brad Cox ja Tom Love työskentelivät ITT Corporation -nimisessä yrityksessä vuoden 1980 alussa. Cox muistelee haastattelussa, että hän sai ensimmäisen esikääntäjän toimimaan vuoden 1982 tienoilla.[2] Vuosi 1983 on merkitty Objective-C:n syntymävuodeksi[41]. Samaan aikaan Stroustrup oli kehittämässä C++-kieltään. Cox ja Stroustrup myös tapasivat tähän aikaan, sillä Strousturp kutsui Coxin käymään hänen Bell Laboratories -yrityksessään. Ohjelmointikielet ovat siis kehittyneet rinnakkain. Coxilla ei ollut aikaisempaa kokemusta Smalltalk -kielestä, joka oli juuri julkaistu. Kaiken tiedon hän sai lehdestä nimeltä Byte, sekä kanssakäymisestä Smalltalkin kehittäjien kanssa.[2] Hänen tarkoituksensa oli saada mahdollisimman paljon Smalltalk-kielen ominaisuuksia Objective-C-kieleen. Hän myös painottaa haastatteluissa, ettei ollut ainut henkilö, joka oli mukana kehittämässä alkupe-

räistä Objective-C -kieltä. Tom Loven lisäksi mukana Objective-C:n kehittämisessä oli muita henkilöitä. Cox mainitsee esimerkiksi, että Steve Naroff on kehittänyt protokollat Objective-C-kieleen. [2]

Tähän mennessä Applen sovellukset ovat tehty Objective-C -kielen avulla, ja Applen tarjoamat kirjastot ovat myös kirjoitettu sillä. Myös LLVM-kääntäjä on toteutettu käyttäen Objective C:tä. Kieli alkaa kuitenkin olemaan vanha, ja modernit ohjelmointikielet omaavat paljon ominaisuuksia mitä Objective-C -kielessä ei ole. Tämä on varmasti ollutkin yksi syy siihen, miksi Apple on kehittänyt Swift-kielen.

2.7 Cocoa Touch

iOS-sovelluskehittämisessä on käytössä Applen tuottama Cocoa Touch -kirjastokokoelma, joka sisältää tärkeitä kirjastoja, joita Apple suosittelee käytettävän. Apple kehottaa käyttämään mahdollisimman korkean tason kirjastoja, ja Cocoa Touch on korkeimmalla tasolla. Tasoja iOS-arkkitehtuurissa on neljä kappaletta, jotka ovat korkeimmasta matalimpaan Cocoa Touch, Media, Core Services ja Core OS[1]. Kehittäjän ei yleensä tarvitse käyttää matalamman tason kuin Cocoa Touch-kerroksen kirjastoja, mutta joissain monimutkaisemmissa tilanteissa voi kehittäjä käyttää matalamman tason kirjastoja.

Cocoa Touch-kirjastoissa on toteutettu asioita korkealla tasolla. Esimerkiksi AirDrop on Applen tapa jakaa tiedostoja kahden sen valmistaman laitteen välillä. Moniajo on myös mahdollistettu korkealla tasolla Cocoa Touch-kirjastoissa. Laitteen ulkopuolelta eli verkon yli tulevat ilmoitukset sekä paikalliset ilmoitukset voidaan käsitellä myös Cocoa Touch-kirjastojen avulla. Eleiden tunnistaminen kuten kosketTaminen, hipaisu, pyyhkäisy jne., joita käytetään kosketusnäytöissä, ovat toteuttettu Cocoa Touch-kerroksessa. iOS-sovelluksia kehitettäessä on käytössä Auto Layout ja Storyboard, joiden avulla käyttöliittymä toteutetaan. Auto Layout huolehtii käyttöliittymän olevan samanlainen eri kokoisilla näytöillä, mikäli rajoittimet ovat asetettu oikein. Storyboard taas esittää kaikkia näkymiä mitä sovelluksessa on olemassa yhdessä näkymässä. Storyboard tiedostoa voidaan muokata Xcoden avulla.[18]

Media-kerroksessa on toteutettu äänet, kuvat ja videot sekä animaatiot ja tietokonegrafiikkaan liittyvät kirjastot kuten OpenGL ES ja GLKit[3] Kehittäjä käyttää Media-kerroksen kirjastoja kun haluaa itse määritellä miten ääntä ja kuvaa käsitellään.[27]

Core Service -kerros tarjoaa käyttöjärjestelmä- eli iOS-palveluiden käyttöön tarkoitettuja kirjastoja. Näiden avulla voidaan käsitellä palveluita kuten säikeistäminen, tiedon kryptaaminen sekä SQLite[4].[20]

Core OS -kerros on matalimman tason kerros iOS-arkkitehtuurissa. Sen tarjoamat

[3]Kirjastojen esitteleminen on tämän diplomityön laajuuden ulkopuolella.
[4]SQLite on relaatiotietokannan hallintajärjestelmä.

kirjastot liittyvät raudan tarjoamiin palveluihin kuten Bluetooth, kiihtyvyysanturit, sormenjäljen tunnistaminen sekä joukko käyttöjärjestelmän tuottamia palveluita kuten levyoperaatiot, muistinhallinta jne.[19].

2.8 ARC

Swift-ohjelmointikielessä on käytössä automaattinen viitteiden laskemiseen perustuva muistihallinta ARC. ARC tulee sanoista Automatic Reference Counting. Automaattinen tässä yhteydessä tarkoittaa sitä, että kääntäjä lisää **retain** ja **release** komentoja aina kun muuttujaan/vakioon viitataan. **Retain** lisää laskurin arvoa yhdellä ja **release** vähentää arvoa yhdellä. Mikäli laskurin arvo on nolla, voidaan muisti vapauttaa koska siihen ei viitata enää.

ARCin ongelma on sykliset viittaukset. Tilanne missä A viittaa B:hen ja B viittaa A:han ei tule koskaan vapautetuksi. Laskurit eivät pääse nollaan syklisen viittauksen vuoksi. Ongelma voidaan ratkaista muuttamalla toinen viittaus heikoksi. Swift-ohjelmointikielessä on kahden tyyppisiä heikkoja viittauksia. **Weak**-tyyppinen viittaus ei voi olla vakio ja se on aina optionaalinen[5]. **Unowned** voi olla joko muuttuja tai vakio, mutta se ei voi olla optionaalinen.

ARC-järjestelmän yhteydessä puhutaan vahvoista ja heikoista viittauksista. Mikäli muistiin ei ole kuin heikkoja viittauksia, voidaan se vapauttaa, vaikka laskurin arvo olisi eri suuri kuin nolla. Vahvat viittaukset siis vaikuttavat vain muistin vapauttamiseen eli mikäli on olemassa yksikin vahva viittaus, ei muistia vapauteta. Palataan alussa esitettyyn esimerkkiin. Mikäli A:n tai B:n viittaus toiseen olisi heikko, tulisi muisti vapautetuksi ARC-järjestelmällä. On sovelluskehittäjän vastuulla, ettei vahvoja viittaussyklejä muodostu sovellukseen, ja näin ollen muistia jää vapauttamatta.[8; 17]

2.9 Kohti Swift-ohjelmointikieltä

Apple julkisti Swift-kielen kesällä 2014. Tämä tarkoittaa sitä, että nykyisin kehittäjillä on mahdollisuus valita Objective-C-kielen ja Swift-ohjelmointikielen välillä. Swift-ohjelmointikieli on kehitetty, jotta iOS-sovelluksia voitaisiin kehittää modernilla kielellä. Objective-C-kielen puutteet alkoivat näkyä, ja syntaksi oli vanha ja ei kovin pidetty verrattuna Swift-ohjelmointikieleen[38]. Syntaksista on tehty helpompi kirjoittaa: mm. funktioita kutsutaan pistenotaation avulla eikä kulmasulkeilla kuten Objective-C-kielessä [15]. Kieleen on otettu mukaan funktionaalisia ominaisuuksia, joiden tarkoitus on antaa kehittäjälle enemmän mahdollisuuksia ohjelmointiin. Kielestä on myös pyritty tekemään turvallisempi kuin Objective-C-kielestä. Yksi

[5]Optionaalisuus esitellään kohdassa 3.2.5.

esimerkki tästä on se, että muuttujat määritellään **var**-sanan avulla ja vakiot **let**-sanalla. Kääntäjä antaa virheen, jos muuttujaa ei muuteta ja suosittelee sen määrittelemistä vakioksi. Kääntäjä huomaa enemmän asioita Swift-ohjelmointikielen tapauksessa, koska Swift-ohjelmointikieli on vahvasti tyypitetty.

Swift-ohjelmointikieli on aloitettu puhtaalta pöydältä, ja Applen tarkoitus oli tehdä paras ohjelmointikieli mitä maailmassa on[15].

3. SWIFT

Apple julkisti Swift -ohjelmointikielen kesällä 2014. Se otettiin innolla vastaan, sillä se tarkoitti sitä, ettei Objective-C -kielen käyttö ollut enää pakollista. Swift-ohjelmointikielestä on tähän mennessä julkistettu kolmas versio ja siitä on myös tehty avoin. Kaikki kehittäjät voivat osallistua päätöksiin siitä, kuinka kieltä kehitetään eteenpäin. Luvussa käydään läpi Swift-ohjelmointikielen uudet ominaisuudet verrattuna Objective-C-kieleen, koska se on ollut ennen ainut vaihtoehto. Luvussa käydään myös jonkin verran ominaisuukaia mitä kummassakin kielessä on mutta pääpaino on kertoa miten Swift-ohjelmointikieli toimii, eikä miten ne erovat toisistaan. Luku perustuu lähteisiin[39; 6; 15; 34; 10; 11]

3.1 Abstraktiotason erot

Luokat, tietueet ja enumit ovat Swift -ohjelmointikielen perusrakennuspalikoita, joiden avulla jäsennetään ja luodaan rakennetta lähdekoodiin. Nämä ovat aseita kompleksisuutta vastaan: data-abstraktio ja tietoabstraktio. Luokan tehdyt instanssit ovat viitetyyppisiä, kun taas tietueiden ja enumien avulla luodut instanssit ovat arvotyyppisiä. Ero on siinä, kuinka muistia hallitaan ja miten kopiointi tapahtuu. Viitetyyppinen instanssi annettuna parametrina, paluuarvona, sijoitettu jne. kopioidaan vain muistiosoite, missä instanssi on. Arvotyyppisen instanssin sijoittaminen toiseen luo sen sijaan kopion. Tämä on merkittävä ero ja tulee ottaa huomioon mietittäessä mikä näistä kolmesta vaihtoehdosta olisi sopiva mihinkin käyttötilanteeseen.

Luokat ovat ”edistyneitä” siinä mielessä, että niillä pystyy tekemään eniten asioita liittyen Swift -ohjelmointikielen ominaisuuksiin. Merkittävä ero luokkien ja tietueen välillä on, että luokkia voidaan periyttää ja tietueita ei. Protokollien[1] avulla kumpiakin voi laajentaa kuten myös enum-tietotyyppejä. Mitä tahansa voi laajentaa; jopa perustietotyyppejä kuten Int ja Double. Olioilta tyyppi, eli mistä luokasta se on luotu, voidaan tarkistaa ajon aikana. Tämä ei ole mahdollista tietueen instansseille. Luokilla on olemassa rakentaja, jonka vastuulla on luomisen yhteydessä saattaa se oikeaan tilaan. Myös tietueille on mahdollista toteuttaa rakentajia. Purkajan vastuulla on vapauttaa resurssit ja tehdä muut mahdolliset siivoustoimet ennen kuin instanssi lakkaa olemasta. Luokille on mahdollista kirjoittaa purkajia, mutta

[1] Kohdassa 3.2.2 on esitelty protokollien idea.

tietueille tämä ei ole mahdollista. Luokan instanssit ovat viitetyyppisiä, ja tämän seurauksena muistinhallinta niiden kohdalla on toteutettu ARC[2]:in avulla.

Tietueita tulee monessa kielessä käsitellä kokonaisina yksikköinä. Swift -ohjelmointikieli mahdollistaa yksittäiseen muuttujaan sijoittamisen, eikä sitä käytettäessä tarvitse luoda uutta alkiota tietueesta, joka sijoitettaisiin kokonaisuudessaan edellisen tilalle. Tämä helpottaa asioita varsikin silloin, kun on olemassa sisäkkäisiä tietueita. Tällöin tulisi muuten rakentaa koko tietuehierarkia uudestaan, jotta yksi arvo saataisiin muutettua. Sisäkkäisen tietueen yhden muuttujan muuttaminen sijoittamalla vain mutettava arvo:

```
struct Mitat {
    let pituus: Int
    let leveys: Int
    var paino: Int
}

struct Laatikko {
    var mitat: Mitat // var, muuttuja
    let numero: Int  // let, vakio
}

var laatikko = Laatikko(
        mitat: Mitat(pituus: 12, leveys: 12, paino: 0),  numero: 1)

laatikko.mitat.paino = 12
```

Edellä olevassa esimerkissä **mitat** on muuttuja, koska sen arvoa halutaan muuttaa. Arvotyyppien kohdalla tulee toimia näin, mikäli arvoa halutaan muuttaa. Mikäli **Laatikko** tietue olisi luokka, voisi **laatikko** olla vakio, koska itse viitettä ei muuteta. Vakioviitteen läpi arvon muuttaminen on sallittua. Mikäli **Laatikko** tietotyyppi olisi toteutettu luokan avulla, olisi **laatikko** viittaustyyppinen. Tämä vuoksi sen voisi esitellä vakiona, koska esimerkissä ei olla muuttamassa viittausta toiseen kohtaan vaan arvon viitteen päässä.

3.2 Geneerisyys

Geneerisyys pyrkii ratkaisemaan ongelmaa, jossa kehittäjä joutuu kirjoittamaan samaan koodia useampaan paikkaan. Tarkoitus on, että voidaan tuottaa geneerisiä eli yleisesti käytettäviä kokonaisuuksia. Tämä avulla koodia tuotetaan vähemmän, jonka seurauksena ylläpidettävää koodia on vähemmän, ja sovelluksien kehittäminen on

[2]Automatic Reference Counting.

nopeampaa. Swift -ohjelmointikielessä geneerisyys on toteutettu usean mekanismin avulla, joihin kuuluu mm. protokollat, laajennokset ja tilanvaraajat[3]. Näiden avulla voidaan tuottaa esimerkiksi kokonaisuuksia, mitkä ovat käytettävissä kaikilla tietotyypeillä. Mahdollisuus käyttää geneeristä toteutusta saattaa joissain tilanteissa on rajoitettu joko kehittäjän halutessa näin tai kielestä johtuvista yksityiskohdista. Kehittäjä voi rajoittaa geneerisen toteutuksen toimivuutta protokollien avulla ja mahdollistaa niiden käyttämistä laajennoksien avulla.

3.2.1 Laajennokset

Laajennoksilla voidaan lisätä toiminnallisuutta luokille, tietueille, enumeille ja protokollille. Tämä siis sisältää myös perustietotyypit, joita Swift -ohjelmointikielessä ovat Int, Float, Double, Bool, Character ja String. Verrattuna protokolliin, joissa määriteltiin pohjapiirustus, jonka tulee vahvistamalla toteuttaa vaatimukset, laajennokset eivät omaa nimeä, koska ne laajentavat aina tietyn tietotyypin. On myös mahdollista laajentaa kaikkia tietotyyppejä paikanvaraajien avulla.

Laajennoksilla voidaa tehdä seuraavia asioita:

- Toteuttaa uusi soveltava rakentaja.
- Lisätä laskettavia muuttujia tietotyypille ja instanssille.
- Lisätä funktio tietotyypille tai instanssille.
- Määritellä indeksointi eli []-operaattori.
- Määritellä ja käyttää sisäkkäisiä tietotyyppejä.
- Vahvistaa protokollaa.

Uusi rakentaja tulee olla tyypiltään soveltava. Nimettyjä rakentajia ja uusia purkajia ei voida toteuttaa laajennoksien avulla. Tämä siitä syystä, että nimettyjen rakentajien vastuulla on saattaa kaikki alustettavat muuttujat/vakiot oikeaan tilaan, ja se ei ole mahdollista laajennoksien kautta. Purkajan toteuttaminen laajennoksilla on myös vastaavasta syystä mahdotonta.

Laajennoksien avulla voidaan lisätä laskettavia tietotyyppejä, mutta ei tallennettuja, eikä muuttujan tarkkailua voida lisätä olemassa oleville muuttujille. Muuttujan tarkkailu on toteutettu Swift -ohjelmointikielessä kahden funktion avulla, jotka ovat **willSet** ja **didSet**. **willSet**-funktiota kutsutaan, kun muuttujan arvo on muuttumassa ja parametrina tulee arvo, mihin se muuttuu. **didSet**-funktiossa on olemassa **oldValue** missä on edellinen arvo ennen muutosta. Funktioiden toteutukseen voi

[3]Vapaat suomennokset sanoista: protocols, extensions ja placeholder.

laittaa logiikkaa mitä halutaan suorittaa, kun muutos on tapahtumassa tai tapahtunut.

Uusien funktioiden toteuttaminen laajennoksilla on mahdollista. Jos halutaan vielä muuttaa tietotyypin olemassa olevia muuttujia, tulee laajennos varustaa **mutating**-sanalla. Indeksointi eli [ ]-operaattorin määritteleminen laajennoksen avulla tapahtuu toteuttamalla **subscript**-funktio. Tämän jälkeen voidaan tietotyypin literaalia, muuttujaa tai vakiota indeksoida [ ]-operaattorin avulla. Parametrin ja paluuarvon tietotyypin voi määritelle miksi tahansa tietotyypiksi. Sisäkkäisten rakenteiden luominen on myös mahdollista ja tapahtuu kuten muuallakin.

3.2.2 Protokollat

Protokolla on kuin pohjapiirustus tai vastaava suunnitelma, jonka avulla on tarkoitus rakentaa jotain. Protokolla yksinkertaisuudessaan määrittelee, mitä funktioita tai muuttujia luokan, tietueen tai enumin tulee toteuttaa, jotta se vahvistaa tietyn protokollan. Sanaa vahvistaminen käytetään, kun kaikki protokollan vaatimat asia ovat täytetty, funktiot toteutettu ja muuttujat ovat olemassa.

Protokolla voi vaatia joko jäsenfunktioita tai -muuttujia. Tämä lisäksi on myös mahdollista vaatia instanssifunktioita tai -muuttujia. Muuttujien yhteydessä vaatimukset ovat tietotyyppi ja nimi. Protokollan avulla voidaan määritellä, onko muuttujan arvo asetettavissa vai vain luettavissa. Muuttujien tapauksessa vaatimuksen voi täyttää tallennetun tai lasketun muuttujan avulla. Laskettu muuttuja ei pidä tilaa vaan on funktio, mikä suoritetaan ja joka palauttaa arvon. Tilavuus saadaan laskemalla leveys kertaa korkeus kertaa syvyys, näin ollen voidaan luoda laskettu muuttuja nimeltä **tilavuus**:

```
struct Laatikko {

    var leveys = 0.0
    var korkeus = 0.0
    var syvyys = 0.0

    var tilavuus = levyes * korkeus * syvyys // Laskettu muuttuja
}
```

Esimerkissä **tilavuus** on laskettu arvo. Sen arvoa ei pidetä yllä, eikä siihen voi sijoittaa mitään. **Tilavuus** palauttaa arvon, joka saadaan laskemalla **leveys**-, **korkeus**- ja **syvyys**-muuttujan kertotulos. Muuttujan **tilavuus** arvo on siis funktio.

Vaadittavien jäsenfunktioiden tapauksessa määritellään vain minkälainen funktion tulisi olla, eli funktion nimi, parametrit ja mahdollinen paluuarvo. Itse toteutusta protokollaan ei tule, koska on jokaisen protokollan vahvistavan asia toteuttaa

vaadittu(t) funktio(t). Protokollassa vaaditut funktiot voivat myös muuttaa vahventavan luokan olemassa olevia muuttujia. Tämä kumminkin vaatii, että funktio määritellään protokollaan **mutating**-sanan avulla[4]. Mikäli protokolla on tarkoitettu käytettäväksi luokkien yhteydessä, ei **mutating**-sanaa tarvitse käyttää erikseen funktioiden määrittelyn yhteydessä, koska kaikki funktiot voivat muokata luokan/olion tilaa.

Uusien rakentajien esitteleminen onnistuu protokollien avulla. Rakentaja voi olla tyypiltään nimetty rakentaja, soveltava rakentaja tai vikaantuva rakentaja[5]. Kaikki rakentajien vaatimukset tulee määritellä **required**-sanan avulla. Tämä varmistaa, että periyttämisen yhteydessä rakentaja tulee toteutettua, tai se peritään kantaluokalta. Mikäli periytetty luokka haluaa ylikirjoittaa kantaluokan toteutuksen rakentajasta, tulee lisätä vielä **override**-sana periytetyn luokan rakentajaan.

Nimetty rakentaja on lopullinen rakentaja, jonka jälkeen kaikki muuttujat ovat alustettu ja mahdollisen yläluokan rakentaja on kutsuttu. Jokaisella luokalla tulee olla ainakin yksi nimetty rakentaja. Periyttämisen yhteydessä tämä vaatimus saattaa täyttyä kantaluokan nimetyn rakentajan avulla. Soveltavan rakentajan tarkoitus on helpottaa nimetyn rakentajan kutsumista esimerkiksi antamalla joillekin parametreille oletusarvot. Soveltavat rakentajat voivat siis antaa myös jotain ennalta määriteltyjä arvoja, ja niiden tarkoitus on helpottaa rakentajan kutsumista. Rakentajien kutsumiseen ja käyttämiseen on kolme sääntöä.

1. Sääntö 1: Nimetyn rakentajan tulee kutsua kantaluokan nimettyä rakentajaa.
2. Sääntö 2: Soveltavan rakentajan tulee kutsua rakentajaa samasta luokasta.
3. Sääntö 3: Soveltavan rakentajan tulee lopulta kutsua nimettyä rakentajaa.

Säännöt siis määrittelevät, että nimetyn rakentajan tulee olla viimeinen rakentaja, jota kutsutaan luokan sisällä. Tämä tapahtuu siitä syystä, että soveltava rakentaja ei ole vastuussa siitä, tulevatko kaikki muuttujat alustettua oikein, vaan nimetty rakentaja on tästä vastuussa. Säännöt myös mahdollistavat sen, että soveltavat rakentajat kutsuvat toisiaan. Lopulta tulee kuitenkin kutsua nimettyä rakentajaa kuten edellä on selitetty.

Vikaantuvan rakentajan vaatimus protokollassa voidaan täyttää kaikilla kolmella rakentajatyypillä. Vikaantuva rakentaja määritellään, kun voidaan olettaa, että jossain tilanteissa alustaminen ei onnistu. Yleensä asia liittyy ulkoiseen resurssiin tai vastaavaan asiaan. Vikaantuvat rakentajat ovat toteutettu Swift -ohjelmointikielessä optionaalisuuden[6] avulla. Optionaalinen muuttuja voi saada arvon **nil**. Tämä on eri

[4]Mutating sijoitetaan **func** sanan eteen.
[5]Suorat suomennokset: Designated, Convenience ja Failable Initializers.
[6]Kohdassa 3.2.5 on asiasta enemmän tietoa.

asia kun Objective-C-kielen **nil**. Swift -ohjelmointikielessä mistä tahansa tietotyypistä voidaan tehdä optionaalinen. Vikaantuva rakentaja voi alustaa olion tilaan **nil**, mikäli alustaminen ei onnistu.

Protokollat ovat tietotyyppejä siinä missä kokonaisluvut **Int**, liukuluvut **Double** yms. ovat tietotyyppejä. Tästä syystä protokollia voidaan käyttää kuin muitakin tietotyyppejä: parametrina ja/tai paluuarvona, muuttujana tai vakiona sekä säiliön tietotyyppinä.

Koska Swift-ohjelmointikieli on vahvasti tyypitetty, on jokaiselle muuttujalle ja säiliölle määriteltävä alussa joko implisiittisesti tai niin että, Swift päättelee itse tietotyypin. Protokollat mahdollistavat korkeammalla abstraktiotasolla tämän tiedon määrittelemisen. Hienoa tässä on se, että koska protokollien idea on olla pohjapiirustus, niin nyt saadaan juuri halutut ominaisuudet muuttujalle tai säiliön alkioille. Säiliöön **maatila** voidaan sijoittaa alkioita jotka vahvistavat protokollaan **Nimi**:

```
protocol Nimi {
    var nimi: String {get} // Voi vain lukea, ei sijoittaa
}

struct Elain: Nimi {
    var nimi: String
    var uros: Bool
}

struct Kone: Nimi {
    var nimi: String
    var valmistaja: String
    var malli: String
}

var maatila: [Nimi] = [
        Elain(nimi: "kissa", uros: false),
        Kone(nimi: "traktori", valmistaja: "MF", malli: "690")]

for alkio in maatila {
    print(alkio.nimi)
}
```

Saman toiminnallisuuden olisi voinut toteuttaa luokkien avulla mutta, mikä olisi ollut tässä tapauksessa sopiva kantaluokka josta luokat **Elain** ja **Kone** olisi periytetty? Luokat olisi myös pitänyt periyttää ylipäätäänsä, ja tässä yhteydessä olisi

voinut tulla jotain funktioita tai muuttujia kantaluokalta. Tässä tulee hyvin esille se, kuinka protokollien avulla tietty asia voidaan varmentaa, vaikka näillä asioilla ei loogisesti olisi mitään yhteistä. Tietueen **Elain** voidaan käyttää minkä tahansa eläimen esittämiseen, eikä vain kotieläinten. Sama pätee tietueen **Kone**. Protokollia voidaan myös miettiä abstraktien kantaluokkien avulla, mikäli ne ovat tuttuja. Esimerkissä ei ole käytetty luokkia vaan tietueita, koska ei ole tarvetta periyttää vaan voi käyttää protokollia.

Protokollat voivat myös periä toisen protokollan vaatimukset. On siis mahdollista periyttää toisesta protokollasta uusi protokolla, ja lisätä siihen vielä uusia vaatimuksia. Näin voidaan luoda samanlaista hierarkiaa kuin luokkien periyttämisessä luodaan. On myös mahdollista luoda protokolla, jonka voi vahvistaa vain luokka. Syitä tähän voi olla riippuen tilanteesta erilaisia, mutta merkittävä syy tähän saattaa olla halutaanko vahvistavan tietotyypin olevan arvo- vai viitetyyppinen. Luokat ovat viitetyyppisiä, kun taas tietueet ja enumit ovat arvotyyppisiä.

Mikäli ei tiedetä vahvistaako jokin tietyn protokollan, voidaan se tarkistaa **is**- ja **as**-operaattorien avulla. Operaattorien avulla voidaan turvallisesti tarkistaa käsittelyssä oleva alkio. Operaattorit toimivat seuraavasti:

- **is**: Palauttaa **true** mikäli vahvistaa protokollan ja **false** mikäli ei.
- **as?**: Palauttaa optionaalisen arvon, joka on **nil** mikäli protokollaa ei ole vahvistettu.
- **as!**: Pakotettu purkaminen, joka aiheuttaa poikkeuksen mikäli arvo on **nil**.

Protokollia voidaan myös laajentaa. Laajennukset ovat toinen tapa, jolla voi tuottaa geneeristä koodia Swift -ohjelmointikielellä. Laajennoksia käsitellään 3.2.1 kohdassa tarkemmin. Laajennoksien käyttäminen protokollien tapauksessa mahdollistaa oletustoteutuksen protokollan vaatimalle funktiolle tai muuttujalle. Tämä onkin varsin hyödyllistä, jos tiedetään, että monessa tapauksessa näin on mahdollista tehdä. Tällöin jokaiseen vahvistavaan luokkaan ei tarvitse toteuttaa samaa koodia, vaan se voidaan laajennoksella toteuttaa vain yhteen paikkaan. Näin koodista tulee ylläpidettävämpää ja nopeampaa tuottaa. Mikäli tietotyyppi täyttää jo kaikki protokollan vaatimukset, mutta ei ole ilmoittanut vahvistavansa protokollaa, voidaan käyttää tyhjää laajennosta kertomaan, että tietotyyppi vahvistaa protokollan.

3.2.3 Geneeriset funktiot

Geneerinen funktio määritellään samaan tyyliin kuin muutkin funktiot Swift -ohjelmointikielessä. Erona on vain se, että paluuarvot ja parametrit ovat tyypittömiä. Niiden merkkaamiseen ja erottelemiseen käytetään paikanvaraajia[7]. Paikanvaraaja

[7]Vapaa suomennos sanasta placeholder.

sijoitetaan nimen perään < > väliin. Tämä tapahtuu siitä syystä, että kääntäjä tietäisi olla etsimättä paikanvaraajan nimistä tietotyyppiä. Mikäli paikanvaraajalla ei ole erityistä korrelaatiota funktion toiminnon kanssa, on perinteisesti käytetty yhtä isoa kirjainta. Muissa tapauksissa isolla aloitettu kuvaava nimi on hyvä tapa. Yhdellä kirjaimella määriteltyjä paikanvaraajia kutsutaan tyyppiparametreiksi, ja nimellä määriteltyjä nimetyiksi parametreiksi Swift -ohjelmointikielessä. **T** kirjain toimii **vaihda**-funktiossa tyyppiparametrina:

```
func vaihda<T>(inout a: T, inout _ b: T) {}
```

Paikanvaraaja korvautuu ohjelmaa suoritettaessa jollain olemassa olevilla tietotyypeillä. Näin pystytään toteuttamaan funktio ilman, että tiedetään mille tietotyypille sen halutaan toimivan. Yllä olevassa esimerkissä on määritelty yksi paikanvaraaja **T**. Näin ollen muuttujien **a** ja **b** tulee olla samaa tietotyyppiä, olivat ne mitä tahansa. On mahdollista määritellä useampi paikanvaraaja, jolloin voidaan antaa geneeriselle funktiolle useampaa eri tietotyyppiä olevia parametreja. Tämä tapahtuu pilkulla erottelemalla < > väliin.

3.2.4 Geneeriset tietotyypit

Voidaan määritellä itse geneerinen luokka, tietue tai enumi. Kuten funktioiden tapauksessa mitään tietotyyppejä ei määritellä valmiiksi, vaan niiden kohdalla käytetään paikanvaraajia. Geneeristen tietotyyppien tapauksessa paikanvaraajat ovat yleensä nimettyjä, koska ne liittyvät vahvasti siihen, mitä tietotyypissä tapahtuu. Näin ollen geneeristä tietotyyppiä voi käyttää millä tahansa olemassa olevalla tietotyypillä, ellei sitä ole rajoitettu. Geneeristen tietotyyppinen laajentaminen on myös mahdollista laajennosten avulla. Tässä tapauksessa voidaan käyttää nimettyjä paikanvaraajia, joita laajennettavassa geneerisessä tietotyypissä on esitelty.

Vaikkakin geneeristen toteutuksien idea on tuottaa toteutus mahdollisimman monelle, on niiden rajoittaminen joissain tilanteissa hyödyllistä. Geneerisen toteutuksen tekeminen kaikille tietotyypeille on melkoinen haaste, joten on hyödyllistä rajoittaa saa joihinkin asioihin kuten tulee olla yhtäsuuruus operaattori määritelty tai tulee olla säiliö. Tällöin voidaan toteuttaa geneerinen toteutus kaikille, jotka täyttävät nämä ehdot. Edellä mainitut olivat vain esimerkkejä, ja Swift-ohjelmointikielessä geneeristen tietotyyppinen rajoittaminen on toteutettu luokkien ja protokollien avulla. Näin ollen kehittäjällä on vapaat kädet luoda protokolla ja rajoittaa geneerisiä tietotyyppejä niin kuin parhaaksi näkee. Tämän vuoksi geneeriset tietotyypit yhdistettynä protokollien rajoituksiin ovat todella ilmaisuvoimaisia ja mahdollistavat paljon kehittäjän näkökulmasta.

Protokollien yhteydessä voidaan käyttää assosiatiivista tietotyyppiä[8] samaan tarkoitukseen kun geneeristen funktioiden ja tietotyyppien yhteydessä käytettiin paikanvaraajia. Kun määritellään protokollaa, ei aina voida tietää, minkä tyyppiselle tietotyypille se otetaan käyttöön. Tällöin määrittelyssä voidaan käyttää assosiatiivista tietotyyppiä tämän tilalla.**Laatikko** protokollassa ei haluta tyyppisiä funktion sijoita parametrit ovat, joten on määäritelty assosiatiivinen tietotyyppi **Alkio**:

```
protocol Laatikko {

        associatedtype Alkio
        mutating func sijoita(item: Alkio)
}
```

Vahvistava luokka määrittelee toteutuksen yhteydessä, mikä on **Alkio** tietotyyppi. Swift -ohjelmointikielessä on olemassa **typealias** eli sana, jonka avulla voidaan eksplisiittisesti kertoa, mikä tietotyyppi tulee olemaan. Tämä ei kuitenkaan ole tarpeen kaikissa tapauksissa, koska Swift -ohjelmointikieli osaa päätellä toteutuksesta minkä tietotyypin **Alkio** saa. Assosiatiivisille tietotyypille voidaan myös asettaa rajoituksia **where** -sanan avulla. Tämän avulla voidaan toteuttaa parempia rajoituksia geneerisiä tietotyyppejä varten. Parempia sen vuoksi, että rajoittamalla mitä tietotyyppejä hyväksytään, voidaan luoda tarkempia ja ylipäätään toimivampia rajoittimia. **where** -sanan avulla voidaan esimerkiksi vaatia, että assosiatiivisen tietotyypin tulee olla **Equatable**, minkä seurauksena sille on määritelty operaattorit ==- ja **!**=.

3.2.5 Optionaalisuus

Kaikki Swift -kielen tietotyypit ja kaikki tietotyypit, joita siihen on itse määritellä, voidaan esitellä optionaalisena. Tämä onkin hyvin tärkeä seikka, koska sama mekanismi toimii kaikille. Optionaalisuuden tarkoitus on tuoda turvallisuutta ohjelmointiin ja löytää mahdolliset virheet jo käännösaikana. Sivutuotteena siitä tulee myös siistimpää lukea ja kirjoittaa, sekä ylläpidettävämpää, koska koodia tulee vähemmän. Ylläpidettävyys syntyy siitä, ettei enää tarvitse luoda erillistä **Bool**-muuttujaa kertomaan onko jokin muuttuja oikeassa arvossa vai ei:

```
var optionaalinen: Int? = nil // Optinaalinen, voi saada arvon nil
// koodia
optionaalinen = 12
// koodia
optionaalinen = nil
```

[8]Vapaa suomennos sanasta associated type.

Optionaalisuus toimii seuraavasti: jos muuttuja[9] määritellään tietotyypiltään optionaaliseksi voi se saada arvon **nil**, eli ”ei mitään” tai ”tyhjä” (miten asiaa haluaa ajatella). Tärkeää on siis huomata, että muuttuja saa jonkin arvon, joka ei ole mikään arvo sen ”normaalilta” arvoalueelta. Esimerkiksi **Int**-tietotyyppi esittää kokonaislukuja. Mikäli luodaan muuttuja, jonka tietotyyppi on optionaalinen **Int** eli **Int?**, voidaan sen arvoksi asettaa **nil**. Ilman optionaalisuutta on käytetty negatiivista arvoa, nollaa tai maksimiarvoa, jonka avulla on yritetty havainnollistaa, että muuttuja ei ole missään oikeassa arvossa. Tämä on kumminkin virheherkkää, eikä kovin helposti ylläpidettävää verrattuna optionaalisuuden **nil**-arvoon:

```
var optionaalinen: Int? = 12
print(optionaalinen!) // 12
// koodia
optionaalinen = nil
print(optionaalinen!) // Käännösaikainen virhe
```

Optionaalisen muuttujan arvo voidaan selvittää joko kiltisti (sovellus ei kaadu mikäli arvo on **nil**) tai pakottaa, jolloin ajonaikainen virhe tapahtuu ja poikkeus heitetään. Onkin järkevää käyttää pakotettua tapaa vain, jos on aivan varmaa, ettei muuttuja missään tapauksessa ole arvossa **nil**. Toisaalta miksi se olisi alun perinkään luotu optionaaliseksi, jos se ei koskaan saa arvoa **nil**. Optionaalisuuden tarkistamiseen kiltisti on useampia tapoja, joiden avulla voidaan selvittää ja purkaa muuttuja tavalliseksi. Voidaan esimerkiksi käyttää ehtorakennetta **if** yhdistettynä vakion esittelyyn[10]. Mikäli muuttujan arvo ei ole **nil**, sijoitetaan se uuteen vakioon ja se on käytettävissä **if**-lohkon sisällä. Toinen vaihtoehto on **guard**-lauseen käyttäminen, joka on ikään kuin käänteinen **if**-lause. Mikäli uuden vakion esittelyyn ei ole tarvetta, voidaan tarkistaa vain arvon olevan eri suuruinen kuin **nil** ja käyttää pakottamista lohkon sisällä:

```
if let eiOptionaalinen = optionaalinen {
        print(eiOptionaalinen) // muuttuja on tavallinen Int
}
// tai
guard let eiOptionaalinen != nil else {
        print(error)
}

// muuttujaa voidaan käyttää määrittelyn ulkopuolella
print(eiOptionaalinen)
```

[9] tai vakio mutta tätä käytetään harvoin koska vakion arvo ei muutu koskaan.
[10] myös muuttujan esittely on mahdollista.

3.3 Funktionaalinen ohjelmointi

Imperatiivinen ohjelmointi perustuu siihen, että kaikella on tila. Tilaa voidaan muuttaa ja se voidaan lukea. Muuttujalla on arvo, mitä voidaan muuttaa. Oliolla on oma sisäinen tila. Funktionaalisessa ohjelmoinnissa ei ole mitään tilan käsitettä, on vain funktioita, jotka saavat syötteitä ja tuottavat tuloksia. Funktionaalisessa ohjelmoinnissa on vakioita ja literaaleja.

Funktionaalinen ohjelmointi on ollut pitkään vain tutkijoiden käyttämä paradigma, jota ei ole käytetty teollisuudessa. Tämä on jossain määrin vieläkin totta, sillä puhtaat funktionaaliset kielet eivät ole saavuttaneet laajaa suosiota, eikä niitä ole otettu käyttöön teollisuudessa sovelluksien kehittämisessä. Swift -ohjelmointikieli tarjoaa imperatiivisen ohjelmointikielen, missä on mahdollista käyttää funktionaalista ohjelmointia. Tämä saattaa olla se yhdistelmä, jonka avulla teollisuudessa voidaan tulevaisuudessa siirtyä kohti funktionaalista ohjelmointia.

Vuonna 1977 John Backus sai ACM Turing Award -palkinnon työstään Fortran-ohjelmointikielen kehittämisessä. Palkinnon saaja pitää yleensä puheen tai luennon palkinnon vastaanottamisen yhteydessä. Backusin palkintopuheen sisältö oli yksikertaisuudessaan se, että funktionaaliset kielet ovat ylivertaisia verrattuna imperatiivisiin. Puheen pitäjä sai palkinnon imperatiivisen kielen kehittämisestä. Tästä voidaan nähdä kuinka paljon parempina moni tutkijakin pitää funktionaalisia kieliä verrattuna imperatiivisiin.

Swift -ohjelmointikielessä on sulkeumia[11], jotka ovat ensimmäisen luokan kansalaisia. Tämä tarkoittaa, että niille/niillä voidaan tehdä samat asiat, mitä muillakin tietotyypeillä voidaan tehdä: esimerkiksi antaa parametrina, olla paluuarvona, tallentaa muuttujaan/vakioon jne. Swift -ohjelmointikielessä on olemassa kolmen tyyppisiä sulkeumia[12], joista yksi on nimetyt eli funktiot. Tämä on erityisen tärkeä asia, koska ongelma imperatiivisten kielen funktionaalisen ohjelmoinnin tukemisessa on ollut se, että sulkeuma ei voi olla funktion paluuarvo. Toinen on funktioiden sivuvaikutukset eli tilanteet, missä funktio muuttaa parametrin arvoa, tai muuta globaalia instanssia. Mikäli parametrina on arvotyyppinen instanssi, ja ei käytetä **inout** -määrettä ei ongelmia funktion kutsumisessa tule. Globaalit muuttujat ovat tosin edelleen ongelma.

Funktionaalisessa ohjelmoinnissa ei ole muuttujia vaan vakioita ja literaaleja. Swift -ohjelmointikielessä on mahdollista käyttää **let** ja **var** -määreitä, joiden avulla määritellään, onko instanssin vakio vai muuttuja. Swift -ohjelmointikieltä käytettäessä onkin suositeltavaa käyttää aina **let** -määrettä kun on mahdollista. Tämä on askel kohti funktionaalista ohjelmointia.[10; 11]

[11] Vapaa suomennosa sanasta Closures.
[12] Lisää kohdassa 3.4.

Perustietotyypit mukaan lukien säiliöt ovat kaikki arvotyppisiä. Näin muuttujat eivät sido osakokonaisuuksia toisiinsa, ellei kyseessä ole olio, jotka ovat viitetyyppisiä. Mikäli osakokonaisuudet ovat irrallisia, kuten funktionaalisessa ohjelmoinnissa ne ovat, on testaaminen ja rinnakkaisuus helpompaa. Tämä onkin yksi syy, miksi funktionaalinen ohjelmointi on ollut nousussa viime aikoina; halutaan hyötyä rinnakkaisuudesta paremmin.

Funktionaalisissa kielissä käytetään paljon hyödyksi laiskaa evaluointia[13]. Sen idea on, että laskutoimitus tehdään vasta kun on aivan pakko. Swift -ohjelmointikielessä on olemassa laiskat tallennetut muuttujat[14]. Näiden avulla voidaan siirtää myöhemmäksi tai jossain tapauksissa jättää kokonaan tekemättä operaatiota. Sanotaan, että meillä on sovellus, jonka avulla voidaan hakea tietoa x ja y verkon yli. Mikäli kummatkin ovat määritelty laiskoiksi, haetaan tieto vasta kun sitä oikeasti tarvitaan, eli käyttäjän sitä halutessa. Perinteisesti kummatkin tiedot olisi haettu kun sovellus käynnistyy.[11]

Se, miksi funktionaalinen ohjelmointi ei ole yleisemmin käytössä, johtuu monesta syystä. Ensinnäkin prosessorit ovat von Neumann -arkkitehtuuriin perustuvia, josta suorana seurauksena on ollut, että imperatiiviset kielet ovat olleet tehokkaampia suorittaa. Tehokkuus on siis ollut alussa ja on edelleen tärkeä tekijä imperatiivisen ohjelmoinnin suosion selittämiseen. Ensimmäiset funktionaaliset kielet eivät olleet samaan aikaan olemassa oleville imperatiivisille kielille kilpailukykyisiä luettavuuden ja tuottavuuden puolesta. Imperatiivisia kieliä käytetään nykyään oikeastaan ainoastaan, ellei ota huomioon tekoälyn sarkaa, missä käytetään LISP-ohjelmointikieltä, joka on funktionaalinen. Tästä syystä melkein kaikki opettelevat ohjelmoimaan jollain imperatiivisella kielellä, jonka seurauksena funktionaalinen kieli saattaa tuntua uudesta käyttäjästä erilaiselta ja vaikealta. Funktionaaliset kielet ovat malli matemaattisille funktiolle. Sekin saattaa olla yksi suuri syy sille, miksi ne eivät ole yhtä houkuttelevia kuin imperatiiviset kielet.[3]

3.3.1 Tuples

Tuplen tarkoitus on olla väliaikainen ja yksinkertainen. Sen avulla voi tehdä asiat kerran tai kaksi, mutta mikäli tulee määritelleeksi samanlaisen tuplen useammin, on syytä miettiä pitäisikö toteuttaa enum, tietue tai jopa luokka. Luokkaa tulisi harkita lähinnä siinä tapauksessa, jos halutaan arvotyypin sijaan viitetyyppinen alkio. Tuple on arvotyyppinen ja kopiodaan:

```
var alkuperainen = (1, 2)
var kopio = alkuperainen
```

[13]Vapaa suomennos: Lazy Evaluation.
[14]Vapaa suomennos: Lazy Stored Properties.

```
alkuperainen.0 = 0

print(alkuperainen) // (0, 2)
print(kopio) // (1, 2)
```

Missä tapauksessa tuplea tulisi sitten käyttää? Yksi parhaista esimerkeistä on funktion paluuarvo. Mikäli halutaan palauttaa useampi arvo, voidaan käyttää tuplea. Tämä on hyvin nopea ja yksinkertainen tapa. Tässä on myös se hyvä puoli, että tuple on olemassa vain siinä kohtaa, eikä se tarvitse erillistä määrittelyä. Jos tuplea ei olisi käytössä, tulisi käyttää tietuetta tai jopa luokkaa. Kummassakin on se ongelma, että ne tulee määritellä, ja Voi olla hieman turhaa määritellä yhden funktion paluuarvoa varten kokonaan uusi tietotyyppi. Tuplet siis auttavat pitämään koodin siistinä ja kompaktina, koska silloin koodiin ei tule ylimääräisiä määrittelyjä.

Paluuarvojen lisäksi tuplea voidaan käyttää funktion parametrinä, jolloin voidaan sitoa yhteen kokonaisuuksia, mikä on ohjelmoinnissa yleinen apukeino koodin jäsentämiseen ja ymmärrettävyyden parantamiseen. Tämän lisäksi voidaan myös vaatia, että saadaan tietty määrä alkioita. Tilanne, missä alkiot normaalisti sijoitetaan taulukkoon, voidaan korvata tuplella. Sanotaan, että halutaan, että on viisi alkiota. Taulukko alustettaisiin tyypillä, mitä alkioiden halutaan olevan, ja tuplen määriteltäisiin sisältävän viisi haluttua tyyppiä. Erona näissä on, että tuple vaatii, että kaikki viisi alkiota tulee olla. Kääntäjä antaa virheen, mikäli niitä on vähemmän tai enemmän. Taulukon tapauksessa voidaan laittaa mikä tahansa määrä alkioita taulukkoon. Tietysti meillä ei ole käytettävissä taulukon operaattoreita, mutta joissain tilanteissa ei tarvita kuin kappalemäärä -vaatimus.

Tuple voi, kuten tietue, sisältää useita arvoja ja ne voivat olla myös mitä tahansa tyyppiä. Jopa funktiot ovat mahdollisia. Funktionaalisessa ohjelmoinnissa käytetään paljon tupleja, joissa on jokin arvo ja tämän lisäksi funktiota, joita kutsutaan jonkin logiikan avulla. Tuple tuo Swift-ohjelmointikieleen funktionaalista ohjelmointia.

Tuplen käyttäminen on siinä mielessä joustavaa, että kaikkia arvoja mitä siihen on määritelty ei tarvitse käyttää tai ns. purkaa muuttujasta/vakiosta. Voidaan ottaa vain ne mitkä halutaan ja käyttää _-merkkiä niiden kohdalla, joita ei tarvita. Tuplen käyttäminen on myös yksikertaista, koska alkiolle ei ole pakko antaa nimeä, vaan niihin voidaan automaattisesti viitata numeroiden avulla alkaen arvosta nolla. Alkioille voidaan antaa nimi joko kaikille tai vain osalle:

```
let tuple = (eka: 1, 2, 3)
print(tuple.eka) // 1

var (yksi, kaksi, _) = tuple
print(yksi) // 1
```

```
print(kaksi) // 2
```

Tuplen käyttäminen lisää turvallisuutta esimerkiksi parametrien tapauksessa. Useamman parametrin ollessa yksi tuple, voi kääntäjä tarkistaa tietotyypit. Mikäli parametrit ovat annettu funktionkutsun yhteydessä väärässä järjestyksessä, palaute tästä tulee vasta ajon aikana, ja palaute on sovelluksen kaatuminen. Paluuarvojen kohdalla ei tarvitse erikseen testata, mitä funktio on palauttanut, jos palautetaan tuplena. Kutsuja tietää mitä odottaa ja voi turvallisesti käyttää funktion palauttamaa tuplea. Optionaaliset tietotyypit toimivat myös tuplen yhteydessä. Tuple voi siis koostua optionaalisista tietotyypeistä. Tuplen avulla voidaan vaihtaa kahden muuttujan arvo päikseen yhdellä rivillä. Tämä lähinnä lisää koodin luettavuutta, mutta joissain tilanteissa myös turvallisuutta. Myös usean muuttujan alustaminen yhdellä rivillä onnistuu tuplen avulla:

```
var tuple = (1, "merkkijono", 2.0, [Int]())
tuple.3.append(12)

print(tuple.0) // 1
print(tuple.3) // [12]
```

3.3.2 Playground

Swift -kielen kanssa samassa yhteydessä Apple julkisti Playground -järjestelmän. Playground -järjestelmän avulla voidaan koodata Swift-kielellä ja nähdä välittömästi mitä koodi tekee. Se on käytössä REPL-järjestelmässä[15], jossa voidaan käyttää Swift-kieltä. Tämän vuoden[16] WWDC-tapahtumassa Apple ilmoitti, että Playground on syksyllä 2016 saatavilla iPad-laitteille.

Playground mahdollistaa paljon asioita ja oikein käytettynä on varsin hyödyllinen työkalu sovelluksien kehittämiseen. Se on erityisen hyödyllinen koodin testaamiseen, jos haluaa tarkistaa jotain. Aikaisemmin täytyi luoda iOS-projekti, alustaa ja määritellä kaikki kuntoon ja lopulta testata miettimäänsä asiaa. Playground vie kaikki ylimääräiset vaiheet pois koodin testaamisesta[17]. Playground on heti valmis, kun tiedosto on luotu. Koska Playground on REPL, niin nähdään välittömästi, toimiiko jokin asia vai ei. Tämä välittömän palautteen saaminen onkin varsin hyödyllistä, sillä on erityisen tärkeää nähdä nopeasti miten asiat toimivat. Kuinka hyödyllistä olisi, mikäli tulevaisuudessa oikeaa sovelluskoodia voisi kääntää ja suorittaa lähes reaaliaikaisesti ohjelmoitaessa? Asioiden testaaminen Playground -järjestelmän avulla myös

[15] Tulee sanoista ”read-eval-print-loop” eli ”lue-evaluoi-tulosta-toista”.
[16] 2016
[17] Testaamisella tarkoitetaan, miten jonkin asia toimii, ei oikeellisuuden testaamista.

estää testikoodien kulkeutumisen/jäämisen sovelluskoodiin. On esimerkiksi mahdollista, että käyttäjä ei jaksa tehdä uutta projektia, koska haluaa nopeasti nähdä, mitä jollain koodinpätkällä tapahtuu, ja selvittää sen upottamalla testattavan koodin oikeaan projektiin. Näin toimittaessa muodostuu ongelmaksi se, että testikoodia saattaa jäädä epähuomiossa oikeaan projektiin.

Visuaalinen palaute siitä, mitä koodissa tapahtuu, auttaa hahmottamaan lähdekoodia. Playground näyttää kaikkien muuttujien arvot, joten enää ei tarvitse miettiä ja muistaa mitä missäkin muuttujassa on kullakin hetkellä. Myös taulukon järjestys on helppo vain vilkaista. Visuaalinen palaute on varsin hyödyllistä opeteltaessa uutta kieltä tai ylipäätään ohjelmoimaan. Playground onkin oiva työkalu ohjelmoinnin opetteluun. Tämä on ollut varmaankin yksi pääsyistä, miksi Apple julkistaa Playground -järjestelmän iPad-alustalle tänä syksynä. Apple on luonut malliksi muutamia oppitunteja siitä, mitä Playground järjestelmällä on mahdollista tehdä. On myös hyvä huomata, että tässä opetellaan käyttämään oikeaa ohjelmointikieltä eikä jotakin pseudokieltä, mitä monet REPL opetusjärjestelmät käyttävät. Lähdenkoodia pystyy muokkaamaan kosketuksen avulla Playground järjestelmässä iPad-alustalla. Tämä on varsin intuitiivista niille, jotka ovat vasta opettelemassa ohjelmoimaan.

3.4 Sulkeumat

C++- ja Objective-C-kielessä on sulkeumia vastaava käsite nimeltä lohko[18]. Swift-kielessä sulkeumat ovat ensimmäisen luokan kansalaisia, joten niillä voi tehdä paljon enemmän.

Funktiot ovat tuttu käsite, joten sulkeumaa ymmärtääkseen kannattaa alussa ajatella sitä funktiona, jolla ei ole nimeä. Sulkeumilla on kolme ominaisuutta, ja niitä on kolmea eri tyyppiä. Näiden kolmen ominaisuuden ymmärtäminen on tärkeää, jotta sulkeumia osaa käyttää oikein. Ominaisuudet ovat:

- On ensimmäisen luokan kansalainen.
- Voi kaapata muuttujan arvon oman näkyvyysalueensa ulkopuolelta.
- Funktio on sulkeuman erikoismuoto.

Ensimmäisen luokan kansalaisuus tarkoittaa, että sulkeumilla voidaan sijoittaa muuttujaan, antaa parametrina funktiolla ja palauttaa paluuarvona. Tämä mahdollistaa niiden säilömisen, ja mikä tärkeämpää, siirtämisen paikasta toiseen. Tämä on tärkeää, koska funktionaalinen ohjelmointi käyttää tätä ominaisuutta paljon hyödyksi. Mikäli sulkeuma sisältää muuttujan, johon tallennetaan parametrina saatu arvo,

[18]Vapaa suomennos sanasta Blocks.

säilyttää se arvon, vaikka sulkeuma tulee suoritetuksi loppuun. Seuraavalla kerralla arvo on sama kuin mitä se edellisellä kerralla oli suorittamisen lopuksi. Funktio on sulkeuman yksi muoto. Se on sulkeuma, jolle on annettu nimi. Sulkeumia on kolme eri tyyppiä ja niiden ominaisuudet ovat:

- Globaaleja, nimellisiä, eivät pidä tilaa yllä.
- Sisäkkäisiä, nimellisiä, voivat kaapata ylemmän tason muuttujien tilaa.
- Paikallisia, nimettomiä, voivat kaapata ympärillä olevien muutujien tilaa.

Listan ensimmäinen kohta kertoo sulkeuman näkyvyysalueesta. Paikallisella tarkoitetaan, että näkyvyysalue muuttuu, koska näissä tilanteissa sulkeuma on tallennettu muuttujaan, jolloin riippuen missä sitä käytetään, on näkyvyysalue siellä. Käytännössä sen näkyvyysalue rajoittuu sen itsensä sisälle. Sulkeuma voi olla nimetty ilman, että se on funktio. Erona näissä on parametrien ja/tai paluuarvon tietotyyppinen määritteleminen. Funktiolle tulee määritellä nämä, mutta sulkeumalle tämä ei ole pakollista mikäli kääntäjä osaa määritellä ne. Nimettömät muuttujat ovat varsin käytännöllisiä, kun halutaan antaa funktiolle sulkeuma parametrina. Nimettömiä sulkeumia käytetään säiliöiden globaalien sulkeumien yhteydessä, kuten **.map**, **.filter**, jne. Viimeinen kohta kertoo voiko sulkeuma kaapata muuttujan tilan. Kaikki ylläpidettävät muuttujat ovat viitteitä alkuperäiseen, ja vaikka alkuperäisen muuttujan näkyvyysalue loppuu, pysyy sulkeumassa eheä viittaus eli muuttuja on olemassa niin kauan kuin sulkeuma viittaa siihen.

Listan viimeinen sulkeumatyyppi on mielenkiintoisin, koska sen avulla voidaan tuottaa funktionaalista ohjelmakoodia. Funktionaalisessa ohjelmoinnissa on tyypillistä, että tehdään pieniä funktioita, jotka saavat syötteitä ja palauttavat ne mahdollisesti muutettuina. Nimettömät sulkeumat on äärimmäinen esimerkki pienestä funktiosta, vaikkakin funktio on sulkeuman. Nimettömien sulkeutumien yhteydessä syntaksi ja kirjoitettava koodin määrä on minimoitu. Funktiolla ei tarvitse olla nimeä. Parametrien tyyppejä eikä paluuarvon tyyppiä tarvitse määritellä, mikäli kääntäjä osaa sen päätellä. Parametreille ei tarvitse antaa nimeä, mikäli ei halua. Parametreihin voidaan viitata **$0**, **$1**, jne. Nimettömät sulkeumat kirjoitetaan yleensä juuri siihen kohtaan, missä niitä käytetään tai ne annetaan parametrina jonnekin. Nimettömät sulkeumat määritellään siinä, mistä ne lähtevät eteenpäin, eikä jossain muualla kuten funktiot monesti ovat määritelty/toteutettu. Luettavuudesta voi olla montaa mieltä **$0**, mutta itse näen sen vain yhtenäisenä tapana. Kun nimettömiin sulkeumiin tottuu, niin se on todella käyttökelpoinen ja yhtenäinen tapa tehdä asiat. Näiden kaikkien asioiden summana on, että nimettömiä sulkeumia on nopeampi kirjoittaa kun funktiota, ne ovat siellä missä niitä käytetään ja ne ovat yhtenäisiä keskenään. Tämä taas parantaa koodin luettavuutta, turvallisuutta, ylläpidettävyyttä sekä kirjoittamisen tehokkuutta.

Säiliöllä Swift -ohjelmointikielessä on useita funktionaalisesta ohjelmoinnista tuttuja funktiota käytettävissä. Näitä ovat mm. **map**, **filter**, **reduce**. Muitakin on, mutta nämä kolme käytetyimpiä. Näitä kaikkia kolmea voidaan käyttää antamalla nimetön sulkeuma, jonka perusteella operaatiot suoritetaan. Tämä on todella ilmaisuvoimainen ja elegantti tapa tehdä asioita. Funktion voi myös antaa parametrina, jos on tarvetta tehdä asia monta kertaa; samanlaisia nimettömiä sulkeumia ei kannata kylvää ympäri ohjelmakoodia. Käytettäessä funktioita nimettömien sulkeumien avulla tulee esille sulkeuman käytön hyödyt siinä, missä se on toteutettu.

Applen kirjoittamassa Swift-oppaassa on hyvä esimerkki siitä, kuinka nimettömät funktiot ovat nopeampia kirjoittaa, helpompia lukea ja ylläpidettävämpiä, koska niissä on vähemmän koodia eli vähemmän kohtia missä tehdä virheitä. Esimerkissä on käyty vaiheittain läpi, miksi nimetön sulkeuma saadaan loppujen lopuksi niin pieneksi. Tarkoitus on saada taulukon sisältä käänteiseen aakkosjärjestykseen

```
let nimet = ["matti", "ville", "kalle",]

// 1
nimet.sort({ (s1: String, s2: String) -> Bool in  return s1 > s2 })
// 2
nimet.sort({ s1, s2 in return s1 > s2})
//3
nimet.sort({ $0 > $1 })
// 4 ei sulkeuma
nimet.sort(>)

// funktiolla toteutettu
func kaanna(s1: String, _ s2: String) -> Bool {
        return s1 > s2
}
```

Ensimmäinen versio on se, miten nimettömiä sulkeumia kannattaa aluksi ajatella funktioina, joilla ei ole nimeä. Kummassakin löytyy tietotyypit ja paluuarvot eksplisiittisesti määriteltyinä. Parametreille on annettu myös nimet. Versiossa kaksi on jätetty parametrien ja paluuarvon tietotyyppi(t) pois, koska kääntäjä osaa päätellä ne. Koska sulkeuma palauttaa vain yhden arvon, voidaan **return** jättää pois ja ottaa käyttöön parametrien pikasyntaksi eli **$**-syntaksi. Nolla viittaa ensimmäiseen parametriin, ja yksi jälkimmäiseen parametriin. Neljäs ja viimeinen on lyhyin, mutta se ei ole enää sulkeumalla toteutettu. Tämä on kuitenkin lyhin tapa suorittaa asia. Alimpana sama toiminnallisuus on toteutettu funktion avulla. Vertailtaessa

kolmatta esimerkkiä ja funktion toteutusta voidaan huomata erot: vähemmän koodia ja ydinasia tulee selkeämmin esille nimettömän sulkeuman avulla, kunhan ensin tottuu **$**-syntaksiin.

4. SWIFT-OHJELMOINTIKIELEN LUPAUS

Swift-ohjelmointikieli on täysin uusi, suhtautui moni siihen syystä varauksella. Ensimmäisessä versiossa oli ongelmia, jotka estivät sen kaupallisen käytön, kuten sovelluksien kaatuminen ja kehitysympäristön[1] puutteellinen tuki Swift -ohjelmointikieltä kohtaan[32]. Kielen suorituskyky oli yksi asia mistä moni oli huolestunut, ja se olikin aluksi alhainen verrattuna Objective-C-kieleen.[38; 33; 37]

Nyt Swift-kielestä on olemassa versio 2 ja seuraava versio eli 3 julkistettiin WWDC -konfrenssissa kesällä 2016[9]. Nykyinen versio on jo osoittanut siinä määrin toimivaksi, että sitä käytetään projekteissa. TIOBE -indeksiä voidaan pitää yhtenä esimerkkinä siitä, koska Objective-C-kieli on listan sijalla 14 ja Swift sijalla 15[40][2]. Vuosi sitten Objective-C-kieli oli sijalla neljä. TIOBE -indeksi kertoo kuitenkin vain kuinka suosittu jokin ohjelmointikieli on, eli sen perusteella ei voida päätellä, että noin puolet Apple Store -portaaliin ohjelmistoprojekteista tehtäisiin Swift- ja puolet Objective-C-kielellä.

Tässä luvussa käydään läpi verkosta löydettyjä havintoja siitä mitä mieltä Swift-ohjelmointikielestä ollaan. Monet havainnot siis perustuvat blog-kirjoituksiin ja arikkeleihin mitä yksityisen henkilöt ovat kirjoittaneet. Mukana on myös suosittuja sivustoja kuten **https://www.raywenderlich.com** sekä **http://nshipster.com**.

4.1 Objective-C-kielen käyttäminen

Swift-ohjelmointikieli on vielä niin uusi, että sitä ei olla haluttu syystä tai toisesta käyttää. Objective-C-kielen käyttämisessä on jatkossakin omat hyvät puolensa, mutta Objective-C-kielessä on myös ominaisuuksia, joiden vuoksi se olisi hyvä vaihtaa Swift-ohjelmointikieleen, kunhan se on tarpeeksi kypsä käytettäväksi ohjelmistoprojekteissa.

Vaikka Apple tarjoaa paljon omien kirjastojensa kautta, tulee välillä tilanteita mihin voisi olla olemassa helpompiakin ratkaisuja. Tämä vuoksi on olemassa paljon kolmannen osapuolen kirjastoja, joita käytetään iOS-sovelluksien kehittämisen yhteydessä. Kolmansien osapuolien tarjoamien kirjastojen suuri määrä on yksi hyvä syy jatkaa Objective-C-kielen käyttämistä. Suosituimmat niistä ovat olleet käytössä monia vuosia, minkä vuoksi niitä on testattu myös käytännössä ja niiden voidaan

[1] XCode
[2] May 2016

olettaa olevan toimintavarmoja.

Melkein kaikki kehittäjät, jotka työskentelevät iOS-sovellusta parissa, osaavat Objective-C-kielen, koska se on ollut niin pitkään ainut kielivaihtoehto. Osa kehittäjistä on voinut tietysti aloittaa ohjelmoimisen vasta Swift-ohjelmointikielen julkistamisen jälkeen ja jättää väliin Objective-C-kielen kokonaan. On siis paljon helpompaa löytää osaavia tekijöitä Objective-C-kielen ohjelmistoprojekteihin, kuin Swift -projekteihin. Tämä on huomionarvoista, koska hyviä sovelluksia ei voi tehdä ilman osaavia kehittäjiä.

Kaikki Applen kirjastot on kirjoitettu Objective-C-kielen avulla. Mikäli sovelluksen kehittämiseen käytetään Objective-C-kieltä, on yhteensopivuus Applen kirjastojen kanssa saumatonta. Kirjastojen käyttäminen on myös helpompaa ajatuksen tasolla, koska käytettäessä samaa kieltä oman koodin kirjoittamiseen on helpompi ymmärtää, miksi jonkin asia toimii tietyllä tavalla.

Syntakseista voi olla montaa mieltä, ja ne ovatkin tavallaan mielipidekysymys, mutta jossain määrin niillä on erilainen ilmaisuvoima[7]. Luettavuus ja kirjoitettavuus ovat myös asioita, mitkä ovat suoraan verrannollisia syntaksiin. Objective-C-kieli on saanut hyvin paljon palautetta, koska siinä **[]**-symbolit ovat suuressa roolissa. Monesti tulee tilanteita, missä näitä merkkejä (varsinkin sulkevia) kasaantuu paljon peräkkäin. Merkkijonojen tulostaminen tapahtuu erityisten merkkien avulla, kuten **%s**, **%d**, **%@**. Sen lisäksi, että kehittäjän tulee muistaa millä mikäkin tietotyyppi tulostetaan, tulee ne laittaa myös oikeaan järjestykseen. Mikäli järjestys on tulostettaessa väärä, koko ohjelma kaatuu[38].

Objective-C-kielessä on käytössä erillinen tiedosto määrittelyä ja toteutusta varten. Tämä on ollut vallalla oleva trendi Objective-C-kieltä kehitettäessä, ja sama ominaisuus löytyy samaan aikaan kehitetystä C++-kielestä. Nykyään ollaan kallistumassa enemmän vain yhden tiedoston suuntaan, eli erillistä määrittelevää tiedostoa ylläpitäminen ei nykyään ole enää tarpeen koska käätäjät ovat kehittyneet. Swift -ohjelmointikielessä on vain yksi tiedosto. Vaikkakin määrittelytiedostosta näkee nopeasti mitä rajapinnassa tai vastaavassa on, tuottaa sen ylläpitäminen turhaa työtä. Tiedon esittäminen kahdessa paikassa vaatii aina toisen paikan ylläpitämistä, jos toiseen tehdään muutos. Aikaa, mikä tulisi käyttää toimintalogiikan toteuttamiseen, käytetäänkin koodin ylläpitämiseen.[15; 4; 38]

Sovelluksien epämääräinen kaatuminen on Objective-C-kielellä toteutetuissa sovelluksissa normaalia. Tämä johtuu usein siitä, että kun sovellus lopulta kaatuu, on suoritus eri paikassa kuin missä syy kaatumiselle on. Tämä taas johtuu siitä, että viestin lähettäminen **nil**-osoitteeseen on hyväksyttävää. Tätä toiminnallisuutta käytetään Cocoa Touch-kirjastoissa suunnittelumallina, eli kirjaston toiminta on rakennettu sen varaan. Kehittäjälle tämä aiheuttaa harmia, sillä virheen tapahduttua voi kulua pitkäkin aika, ja sovellus voi ehtiä hyvin kauaksi siitä, missä virhe on

alunperin tapahtunut.[4; 38]

Objective-C-kielen muistinhallinta on kehittäjän päätettävissä riippuen siitä, miten esittelee muuttujan/vakion. Monesti annetaan ARC:in huolehtia muistinhallinnasta, ja näin ollen siihen ei tarvitse kiinnittää erityistä huomiota. ARC:in käyttäminen ei kuitenkaan ole mahdollista matalan tason kirjastoissa, joita on esimerkiksi grafiikkakirjastot. Näitä käytettäessä tulee kehittäjän itse huolehtia muistinhallinnasta. Tämä mahdollistaa virheet, ja niiden seurauksena muistivuodot. Näiden vuotojen selvittäminen on monesti aikaavievää ja hankalaa.[38]

Objective-C-kielessä ei ole olemassa nimiavaruuksia. Asia on hoidettu siten, että jokainen käyttää kolmea kirjainta omien toteutuksiensa edessä. Tämä ei tietenkään ole täysin idioottivarma järjestelmä, koska ei voi olla varma, ettei jokin sovellus käytä kahta eri kirjastoa, joilla sama kolmen alkukirjaimen koodi. Törmäykset ovat väistämättömiä. Apple käyttää itse kahta isoa kirjainta. Tämä erottelee muut Applen koodista, mutta koska muut voivat valita satunnaisesti kolme isoa kirjainta, ei järjestelmä ole aukoton.[38]

Kirjastot ovat staattisia eli ne tulee kääntää sovellukseen mukaan, ja ne kaikki ladataan heti, kun sovellus käynnistyy. Tämä saattaa johtaa tehokkuusongelmiin isojen kirjastojen kanssa tai tilanteissa missä on paljon kirjastoja. iOS-laitteet ovat kuitenkin kannettavia, joten tehokkuus on aina asia mihin tulisi kiinnittää huomiota. Nykyään ihmiset odottavat laitteiden olevan responsiivisia, tai he luulevat sovelluksen olevan jumissa.

Objective-C-kieli on kehitetty C-kielen pohjalta ja onkin siksi sidoksissa siihen edelleen. Objective-C-kieli ei siis voi kehittyä omillaan vapaasti, vaan joitain ominaisuuksia varten C-kielen tulisi kehittyä ensin. On myös hyvä huomata, että Applen julkaistua Swift-kielen on olettavaa, että Apple panostaa sen kehittämiseen paljon, ja on epävarmaa kuinka paljon Objective-C-kieltä tullaan kehittämään eteenpäin. Voidaankin kysyä, oliko Swift-ohjelmointikielen julkistaminen viimeinen naula Objective-C-kielen arkkuun.[38]

4.2 Swift-ohjelmointikielen käyttäminen

Kehittäjillä saattaa olla erilaisia syitä kokeilla Swift-ohjelmointikieltä tai jopa käyttää sitä oikeassa ohjelmistoprojektissa. Moni asia vaikuttaa siihen, kuten edellisessä kappaleessa käsiteltiin, tuleeko jatkaa Objective-C-kielen käyttämistä. Swift on uusi kieli, mistä johtuen moni asia ei ole vielä saavuttanut lopullista muotoaan, ja joitain asioita on tullut esille, kun kehittäjät ovat sitä käyttäneet. Apple ei lupaa, että Swift -ohjelmointikieli olisi taaksepäin yhteensopiva. Kielen kehittäminen ei ole näin lukittu alussa tehtyihin ratkaisuihin. Mikäli alussa on menty suunnittelussa kunnolla metsään, ei koko kieli silti ole tuomittu alussa tehtyjen virheiden vuoksi. Kehittyminen on vapaata vielä tässäkin vaiheessa, vaikka Swift -ohjelmointikieli on julkistettu

kehittäjien käytettäväksi. Toisaalta kolikon kääntöpuolena on, että kehittäjät eivät voi luottaa siihen, että nyt toimiva koodi toimisi tulevaisuudessa. Tämä on yksi syy miksi Swift-ohjelmointikielen käyttäminen ohjelmistoprojektissa saattaa tuntua epäilyttävältä.

Kolmannen osapuolen kirjastoja on nyt kahden vuoden jälkeen tarjolla kohtuullinen määrä, mutta selvästi vähemmän kuin mitä Objective-C-kielellä on olemassa, yksikertaisesta syystä, että Swift -kieli on niin uusi. Tämä saattaa vaikuttaa negatiivisesti Swift-ohjelmistokielen käyttöönottamiseen. Onko kirjastot kirjoitettu oikeasti uudelleen ja käyttäen Swift-ohjelmistokielen uusia ominaisuuksia, vai onko ne vain käännetty Objective-C-kielen kirjastoista Swift-ohjelmointikielelle? Tässäkin yhteydessä tehokkuus saattaa olla ratkaisevassa asemassa.

Kaupallisissa ohjelmistokehityksessä aika on rahaa ja raha ratkaisee. Swift-ohjelmistokielen käyttämisessä onkin monen kehittäjän kohdalla uuden opettelemisen paikka. Joidenkin kehittäjien mielestä uuden oppiminen on mieluisaa, mutta toiset taas saattavat nähdä opettelun pakkona, mikä ei edistä oppimista. Yksilöiden välillä on myös paljon eroja siinä, kuinka nopeasti he pystyvät omaksumaan asioita. Pystyykö kehittäjä päästämään irti Objective-C-kielen ajattelumalleista ja hyväksymään Swift-ohjelmointikielen tavan tehdä asioita? Nämä kaikki tekijät vaikuttavat tuottavuuteen ja edelleen valintaan, otetaanko Swift-ohjelmointikieli käyttöön vai ei.[32]

Swift-ohjelmointikieli on tulevaisuus, johon Apple näyttää panostavan. Voidaan ajatella, että on vain ajan kysymys, koska kielen opetteleminen on edessä, jos haluaa jatkaa alalla. Tietenkään ei voi olla varma tippuuko Objective-C-kielen tuki kokonaan pois iOS-kehittämisestä. Toisaalta kehittäjillä on nyt sauma tulla yhdeksi parhaista Swift-ohjelmointikielen osaajista, koska kenelläkään ei ole ollut paljon aikaa käyttää/opetella Swift-ohjelmointikieltä. Kolmen vuoden päästä on olemassa maksimissaan viiden vuoden Swift-ohjelmointikielen osaajia. Tämä voi olla valttikortti niin yritykselle kuin kehittäjälle itselleen. Olemalla yksi niistä, joilla on eniten osaamista mitä tulee Swift-ohjelmointikielellä ohjelmistoprojektien toteuttamiseen, varmistaa itselleen etulyöntiaseman.

Ensimmäisen vuoden aikana kehittäjiltä tuli palautetta, että Swift-ohjelmointikieli ei ole valmis, koska syystä tai toisesta sovellus saattoi kaatua ja vika oli itse kielessä. Nämä olivat todella hankalia tilanteita kehittäjän kannalta, sillä ratkaisua ongelmaan ei voi tehdä itse, vaan tulee keksiä jonkin tapa, miten tällainen tilanne voidaan kiertää ja odottaa, että Swift-ohjelmointikielen kehittäjät tuottavat ratkaisun. Kääntämiseen kuluva aika on myös tähän mennessä ollut kritiikin kohteena, mutta ilmeisesti tähän on tullut koko ajan parannusta. Tulevaisuus näyttää lupaavalta, koska Apple on kiinnittänyt tähän huomiota koko Swift-ohjelmointikielen olemassaolon ajan. Vertailuja sovellusten keskinäisestä nopeudesta on olemassa, tällä hetkellä Objective-C-kieli on nopeampi verrattuna Swift -kieleen. On olemassa myös

algoritmitestejä siitä, kuinka kauan kummallakin kielellä kuluu suorittamiseen. Näiden vertailuiden yhteneväisyyksistä voidaan olla montaa mieltä, mutta koska Swift-ohjelmointikieli on lähestynyt koko ajan Objective-C-kielen aikoja, voidaan olettaa, että Applen kehittäjät ovat kiinnittäneet tähänkin asiaan huomiota.[33; 37]

Applen kirjastot eli Cocoa Touch-kirjastot, joita käytetään iOS-kehittämisessä, ovat kirjoitettu Objective-C-kielellä. Tämä tuottaa jonkin verran ongelmia kun käytetään Swift-ohjelmointikieltä oman sovelluksen kehittämiseen. Ainut merkittävä ongelma enää on ns. siltaesittelytiedostojen[3] kanssa. Muuten Apple on tehnyt hyvää työtä, ettei enää olisi ongelmia Cocoa Touch-kirjastojen käyttämisessä Swift-projekteissa. Aluksi siltaesittelyiden kanssa saattaa olla ongelmia, mutta kun ne tajuaa ja oppii käyttämään niitä, eivät ne ole niin iso ongelma etteikö Swift-ohjelmointikieltä voisi käyttää niistä huolimatta.[15; 38]

Syntaksi on saanut paljon huomiota, koska siitä on pyritty tekemään mahdollisimman turvallinen, luotettava ja kirjoitettava. Yksi suuri ero ja monelle helpotus on, että puolipiste **;** ei ole pakollinen Swift-ohjelmointikielen rivin lopussa. Kaikki, jotka ovat ohjelmoineet käyttämällä Objective-C-kieltä, ovat varmasti toistuvasti kääntäneet ja huomanneet puolipisteen puuttuvan. Tämä ei ole yksittäisenä tapahtumana kovin suuri asia, mutta jos mietitään kuinka paljon aikaa vuoden, kahden tai kymmenen vuoden aikana on mennyt hukkaan puolipisteen takia, se muuttuukin suureksi asiaksi. Puolipistettä voi käyttää mikäli haluaa, ja sitä tulee käyttää, jos samalla rivillä on kaksi eri suoritettavaa asiaa.[15; 38; 7]

Swift -ohjelmointikielessä on käytössä vain yksi tiedosto eli niinä ei ole erillistä määrittelytiedostoa. Tämä vähentää ylläpidettävää koodia, ja asioita ei ole esitetty turhaan kahdessa eri paikassa. Tämä säästää aikaa ja vähentää virheiden mahdollisuutta, mikä taas lyhentää käännösvirheiden korjaamiseen kulutettua aikaa. [15; 38; 4]

Objective-C-kielessä **nil**-osoitteeseen viestin lähettäminen on hyväksyttävää, ja Cocoa Touch käyttää sitä tarkoituksella joissain tapauksissa. Swift-ohjelmointikielessä taas **nil**-osoittimen käsitteleminen aiheuttaa aina ja välittömästä ohjelman kaatumisen. Tämä on todella hyödyllistä, koska näin voidaan saada tieto siitä, missä ohjelma kaatuu eikä tarvitse ensin etsiä kohtaa missä se on kaatunut, kuten Objective-C-kielen yhteydessä joissain tapauksissa täytyy tehdä. Tämä jälleen kerran lyhentää mahdollisesti aikaa, joka kuluu virheen korjaamiseen.[15; 38; 4]

Muistinhallinta on monesti haastava ja tuottaa vuotoja mikäli kehittäjä itse toteuttaa sen. Tämä vuoksi moni moderni ohjelmointikieli käyttää jotain tapaa huoltaa muistinhallintaa. Swift-ohjelmointikielen tapauksessa ARC pitää huolta muistinhallinnasta jopa alemman tason asioiden yhteydessä. Objective-C-kielen tapauksessa kehittäjän tuli itse huolehtia muistin varaamisesta ja vapauttamisesta.[15; 38]

[3]Vapaa suomennos Bridge Header.

Merkkijonojen interpolointi mahdollistaa lukuarvojen ja muiden symbolien tulostamisen helposti. Tulostamista helpottaa, että interpolointi hoitaa muutoksen esimerkiksi kokonaisluvusta merkkiin tulostamisen yhteydessä. Tulostuslausekkeessa voidaan myös suorittaa laskutoimituksia, joiden tulos tulostetaan ilman tarvetta tallentaa sitä väliaikaiseen muuttujaan/vakioon. Tämä on yksi syy, miksi Swift-ohjelmointikielen käyttäminen verrattuna Objective-C-kieleen on parempi vaihtoehto, koska Objective-C-kielessä ei ole merkkijonojen interpolointia.[15]

Objective-C-kielessä ei ole nimiavaruuksia, joten törmäykset ovat olleet aina iso ongelma. Swift-ohjelmointikielessä kehittäjän ei tarvitse huolehtia nimiavaruuksista, koska ohjelmointikieli pitää siitä automaattisesti huolta. Mikäli lähdekoodit ovat sidottu samaan käännettävään yksikköön, ovat ne saman nimiavaruuden sisällä. Näin ollen kolmannen osapuolen kehittämät kirjastot voivat rauhassa käyttää mitä tahansa nimiä omien funktioittensa yhteydessä ilman, että tarvitsee miettiä tapahtuuko niitä käytettäessä törmäys. Tämä yksinkertaistaa kehittäjän työtä ja vähentää virheitä.[15; 38]

Swift -ohjelmointikieli tukee dynaamisesti ladattavia kirjastoja. Tämä on ensisijaisen tärkeää kielen kehittymisen kannalta, koska kieli voi kehittyä omaa tahtiansa ilman, että tarvitsee odottaa Applen kirjastoilta tulevaa isoa päivitystä kerran vuodessa. Tähän mennessä Objective-C-kielen ja staattisesti linkitettyjen kirjastojen suhteen on ollut niin. Dynaamisesti ladattavat kirjastot tuovat tehokkuutta, koska kaikkia kirjastoja ei ladata kun sovellus käynnistyy, vaan sitä mukaa kun niitä tarvitaan. Tämä saattaa lyhentää sovellusksen käynnistymisaikaa hyvin paljon.[15; 38]

Uusi hieno työkalu Swift -ohjelmointikielen opiskeluun/kokeiluun on Playground[4]. Tämän avulla kehittäjä voi kokeilla Swift-ohjelmointikieltä. Koodi kääntyy automaattisesti ja antaa visuaalista palautetta tapahtumista, joten muuttujien arvoja ei tarvitse koko ajan tulostaa, jos niitä haluaa tietää. Tämä on hyödyllinen ominaisuus kun haluaa kokeilla tai tarkistaa jotain: ei tarvitse tehdä testiprojektia, asettaa sitä kuntoon ja kääntää. [15; 38]

Monessa ohjelmointikielessä on mahdollista kirjoittaa yksi rivi if-ehdon jälkeen, joka suoritetaan vain jos ehto on totta ilman lohkoa. Tämä on vaarallinen ominaisuus, koska kaksi peräkkäistä if-lausetta ilman lohkoa saattavat käyttäytyä kääntäjästä riippuen eri lailla. Tilanteissa missä ei ole lohkoja ja halutaan lisätä toinen rivi tekemään jotain muuta, jää helposti lohko lisäämättä. Sen seurauksena toinen rivi tulee suoritettua riippumatta if-ehdosta. Tämä ei ole Swift -ohjelmointikielessä mahdollista, koska kontrollirakenteita ei voi kirjoittaa ilman lohkoa. [15; 38]

Vahvan tyypityksen vuoksi kääntäjä osaa päätellä monesti minkä tyyppinen instanssi halutaan luokka, mikäli siihen sijoitetaan arvo. On myös muita tilanteita, joissa kääntäjä osaa päätellä tyypin ilman, että sitä tarvitsee erikseen kertoa. Tämä

[4]Suomeksi leikkikenttä.

vähentää koodia, parantaa ylläpidettävyyttä, vähentää virheitä ja nopeuttaa kehittämistä. Kaikki positiivisia asioita ohjelmistoprojekteja mietittäessä.[15; 38]

Objective-C-kielessä ei ole olemassa **tuple**-rakennetta, joka mahdollistaa useamman tietotyypin väliaikaisen yhteen sitomisen kuten paluuarvojen yhteydessä. Paluuarvot eivät ole ainut rakenne missä tästä on hyötyä. Tietueisiin voidaan laittaa **tuple**-rakenteen avulla muuttujia, kun halutaan sitoa ne ajatuksen tasolla yhteen. Swift -ohjelmointikieli tukee **tuple**-rakennetta.[15]

Swift -ohjelmointikieli omaa joitain funktionaalisia ominaisuuksia. Funktionaalinen ohjelmointi on kasvattanut suosiotaan, koska se on rinnakkaisissa järjestelmissä tehokkaampi kuin imperatiivinen ohjelmointi. Funktionaalisen koodin testaaminen on myös helpompaa, koska siinä asiat monesti muodostavat omia pieniä kokonaisuuksiaan, jotka ovat irti toisistaan. Onkin tärkeää, että kun käyttää funktionaalista ohjelmointia tukevaa Swift-ohjelmointikieltä, tuotetaan funktionaalista koodia eikä vain tehdä niin, kuin aiemmin imperatiivisella tyylillä tehtiin. Mikäli näin tehdään, ei saada täyttä hyötyä Swift-ohjelmointikielestä. Tällöin voitaisiin käyttää mieluummin Objective-C-kieltä, jota kehittäjät osaavat paremmin.[15; 34]

Aikoinaan kun C++-kieli yleistyi, ja olio-ohjelmointiparadigma oli suuressa suosiossa, pakotettiin kehittäjiä käyttämään sitä. Tästä tuli ongelma, koska aiemmin C-kieltä käyttäneet kehittäjät eivät osanneet käyttää C++-kieltä, vaan väänsivät väkisin/tahallaan/osaamattomuuttaan C++-koodia, joka oli käytännössä C-koodia koska olioparadigmaa ei ollut käytetty oikein. Tämä sai aikaan vaikeasti ylläpidettäviä projekteja. Onkin siis tärkeä tietää, ovatko kehittäjät valmiita sitoutumaan uuden kielen käyttöönottamiseen ja käyttämään sitä niin kuin sitä kuuluu käyttää.

4.3 Yhdistelmä Objective-C-kieltä ja Swift-ohjelmointikieltä

Apple on selvästi nähnyt vaivaa Objective-C- ja Swift -ohjelmointikielien yhdistämiseen. On mahdollista käyttää molempia ohjelmointikieliä samassa projektissa. Tämä ei kuitenkaan ole aivan niin suoraviivaista, kuin miltä Apple saa sen kuulostamaan. Toiminnassa on eroa, jos Swift-ohjelmointikielellä kirjoitettua koodia käsitellään Objective-C-kielellä ja toisin päin. Kummassakin tapauksessa siltaesittelyt ovat tapa, jonka avulla kommunikointi tapahtuu.

Ensimmäisellä kerralla kun toisella kielellä toteutettu tiedosto lisätään projektiin, ehdottaa XCode siltaesittelyjen luomista. XCode tekee automaattisesti tarvittavan tiedoston ja listaa sinne luokat. Siltaesittelytiedosto on myös mahdollista tehdä halutessaan itse. Siltaesittelyyn tulee listata kaikki luokat, joihin haluaa pääsyn Swift -ohjelmointikielestä.[28] Swift-ohjelmointikielellä toteutettujen luokkien eteen tulee lisätä **@objc**, jotta niitä voidaan käyttää Objective-C-kielellä.

Koska Swift -ohjelmointikieli on kehittyneempi kuin Objective-C-kieli, on olemassa joitain ominaisuuksia mitä ei voida käyttää ristiin. Seuraavat Swift-ohjelmointi-

kielen ominaisuudet eivät ole käytettävissä Objective-C-kielellä, vaikka siltaesittelyt olisi tehty.[28] Nämä ovat geneerisyys, tuple, enum-tyyppi[5], tietueet, type alias, sulkeumat, sisäkkäiset tietotyypit ja funktionaalisuus

Listaa läpikäymällä huomaa, että melko moni asia ei ole enää mahdollista sekaprojektissa. Voidaan myös ajatella, että kummankin kielen huonot puolet tulevat mukaan yhdistettäessä ne samaan projektiin. Ylläolevassa toimimattomien ominaisuuksien listassa on useita ominaisuuksia joiden takia Swift-ohjelmointikieli on modernimpi kuin Objective-C-kieli.[36]

[5]Joilla ei ole int.raw arvoa.

5. KEHITTÄJIEN KOKEMUKSET

Luvussa käydään läpi kolmen haastateltavan näkemyksiä ohjemointikielistä. Jokaisella haastatellulla on iOS-tausta ja vaihtelevasti Objective-C- ja Swift-osaamista. Haastateltavat on valittu, jotta työssä saataisiin katettua kaikki kolme mahdollista tapausta: vain Objective-C-kieltä käyttävä projekti, vain Swift-ohjelmointikieltä käyttävä projekti tai sekaprojekti, jossa käytetään kumpaakin kieltä.

Haastattelut olit vapaamuotoisia vaikkakin haastattelulle oli suunniteltu runko. Runko oli lähinnä sitä varten, että kaikki asiat mitä diplömityönkannalta tuli käsitellä tulee käsiteltyä. Haastattelut äänitettiin ja suoritettiin kahden kesken. Haastattelun purkaminen tapahtui muistiinpanojen ja äänitteen avulla. Kohta perustuu haastatteluihin, ja sen sisältää siksi haastateltavien mielipiteitä joiltain osin.

5.1 Objective-C-projekti

Objective-C-kieli oli ennen Swift-ohjelmointikielen julkistamista ainut vaihtoehto kehittää sovelluksia iOS-käyttöjärjestelmälle. Saarenpänä tausta on mobiili-Linux puolelta, jossa hän on käyttänyt C:tä ja C++:aa ja näiden päällä mm. Qt-kirjastoa. Kaksi vuotta sitten Saarenpää huomasi, että Linuxin mobiilipuoli on menossa alas, ja oli valittava uusi alusta johon suuntautua. Hän valitsi iOS-alustan. Objective-C-kieli oli Saarenpään mielestä suorastaan puistattavan ja hurjan näköistä. Syntaksi ei siis todellakaan miellyttänyt C/C++-taustan omaavaa Saarenpäätä. Objective-C-kieli oli kuitenkin opeteltava, koska Saarenpää oli valinnut iOS-alustan. Syy valintaan oli hänen näkemyksensä, että iOS-kehittäjiä tarvitaan tulevaisuudessa. Objective-C-kieli on kuitenkin ohjelmointikieli muiden joukossa, Saarenpää huomauttaa, ja hän onkin oppinut käyttämään sitä. Kahden vuoden käyttämisen jälkeen mielipide syntaksia kohtaan ei ole muuttunut ja [ ]-operaattori on edelleen Saarenpään mielestä ikävä käyttää ja lukea.

5.1.1 Objective-C-kielen käyttäminen

Objective-C-kieli ei ole sen ihmeellisempi kuin muut vastaavat ohjelmointikielet, kuten C- ja C++-kieli, joita Saarenpää on aiemmin käyttänyt. Kuten muissakin kielissä on Objective-C:ssä omat erikoisuutensa. [ ]-operaattorin käyttäminen on erikoinen tapa kutsua funktioita, ja Saarenpää ei välitä tästä syntaksista. Muutenkaan Objective-C-kielen syntaksi ei saa Saarenpäältä kehuja. Linuxin mobiilipuolella

työskennellessään hän on käyttänyt C- ja C++-kieliä, jotka ovat todella tehokkaita. C-kieli on yksi tehokkaimmista ohjelmointikielistä mitä on olemassa. Voisi jopa sanoa, että C-kieli on kaikista tehokkain. Saarenpää huomauttaa, että peleissä, joissa pitää saada kaikki tehot irti käytetään nimenomaan C-kieltä. Tämä vuoksi Objective-C-kielen tehottomuus on rasittanut häntä koko ajan. Ratkaisuna tähän Saarenpää on käyttänyt C-kieltä joskus suoraan Objective-C-kielen sisällä. Tämä on mahdollista, koska Objective-C-kieli on kehitetty C-kielen pohjalta[35]. Periyttäminen NSObject-luokasta on hänen mielestään vaivalloista. Saarenpää haluaisi luoda oikeasti omia luokkia ja olioita. Tämäkin lähinnä tehokkuuden vuoksi, koska periyttäminen NSObject-kantaluokasta tuo tehottomuutta. Saarenpää kertoo esimerkkinä, että eräässäkin tapauksessa C-kielen käyttäminen oli karkeasti noin kymmenen kertaa nopeampi vaihtoehto kuin Objective-C-kielen käyttäminen. Tapauksessa käsiteltiin kuvadataa, eli pikseleitä käytiin läpi ja muokattiin. Datan ollessa säilöttynä C-kielen avulla oli se siis kymmenen kertaa nopeampi, kuin datan ollessa Objective-C-kielen säiliössä. Nopeuden huomasi, kun sitä käytiin läpi, mitä tehtiin toistuvasti dataa käsitellessä.

Lohkot ovat Saarenpään mielestä Objective-C-kielessä miellyttävämpiä käyttää verrattuna niiden vaihtoehtoon, delegaatteihin. Delegaatit ovat hänen mielestään liian raskaita käytettäviksi syntaktisesti, koska niitä käyttäessä joutuu kirjoittamaan paljon koodia, jotta saa tehtyä asian. Lohkot ovat siis yksinkertainen ratkaisu tähän asiaan, mutta niidenkin käyttämisessä on kääntöpuolensa. Kääntöpuolena lohkojen käytössä on vahvat syklit. Lohkojen yhteydessä näiden vahvojen syklien muodostuminen on todella yleistä, jolloin käyttäjä joutuu miettimään kumpi viittauksista on heikko.

Cocoa Touch-kirjasto on Applen tarjoama kirjasto, jonka avulla kehitetään iOS-sovelluksia. Kirjastolla on pitkä historia, jonka voi huomata myös luokkien nimessä olevista NS-kirjaimista. NS viittaa NEXTSTEP-sanaan, joka edelsi Apple Inc. yritystä[35]. Kirjasto on vanha, ja haastatteluissa ilmeni, että tästä on seurannut ongelmia Saarenpäälle. Kirjastossa on säilytetty yhteensopivuus taaksepäin, mikä on saattanut vaikuttaa siihen, että jotain bugeja ei ole korjattu kirjaston tarjoamista perusrakenteista. Saarenpään turhautuneisuutta kuvastaa hänen kommenttinsa: ”Pitäisi varmaan itse kirjoittaa omat”. Objective-C-kielen säiliöt ovat myös olleet Saarenpään mukaan rasittavia käyttää, koska perustietotyyppejä kuten Int, Double, Float ei voi tallentaa säiliöön, vaan niihin tulee sijoittaa NSNumber-tyyppisiä olioita. Tämänkin ongelman ratkaisuna hän on käyttänut C-kielen tietorakenteita, kuten ratkaisuna tehottomuuteenkin. .

5.1.2 Projekti

Saarenpää on tehnyt kaikki iOS-projektinsa Objective-C-kielellä. Hänen nykyinen projektinsa on myös iOS-projekti, mutta hän toteuttaa siihen palvelinpuolta. Projekti on ollut käynnissä reilun kuukauden. Edellinen projekti, jossa hän on ollut melkein alusta asti mukana, on kestänyt vuoden verran. Projekti on Objective-C-kielellä toteutettu ja sen kehittäminen on edelleen käynnissä.

Projektin asiakas oli itse toteuttanut alun projektille Objective-C-kielellä, ja kun projekti oli ulkoistettu, oli tässä yhteydessä keskusteltu tulisiko vaihtaa Swift -ohjelmointikielelle. Saarenpää ei ollut projektissa alusta asti mukana, joten ei osaa sanoa miksi projektissa oli jatkettu Objective-C-kielellä tai miksi sitä ei ollut vaihdettu Swift-ohjelmointikieleen. Hän kuitenkin arveli, että aika (tai raha) oli ollut ratkaiseva tekijä valinnassa, koska Swift-ohjelmointikieli olisi ollut kaikille projektin toteuttajille uusi kieli. Ohjelmointikielen vaihtamisesta on kuitenkin keskusteltu kevyesti koko toteutuksen ajan, mutta kunnollista keskustelua asiasta ei ole käyty. Tässäkin on ollu syynä se, että projektissa on kiire ja raha on tiukalla.

Saarenpäällä ei ole oikeastaan yhtään kokemusta Swift -ohjelmontikielestä. Hän on kerran kokeillut Swiftiä, mutta ei ole opetellut kieltä sen enempää, koska olosuhteet eivät ole olleet opettelulle suotuisat. Hän haluaisi kuitenkin tulevaisuudessa opetella Swift -ohjelmointikielen. Siihen ei vain ole ollut aikaa tähän mennessä. Vaikka Saarenpää ei osaa Swit-ohjelmointikieltä, aloittaisi hän uuden projektin sillä ennemmin kuin Objective-C-kielellä, jos se olisi projektissa mahdollista. Mahdollisuuden edellytys tässä tapauksessa olisi se, ettei projektissa olisi kova kiire, vaan uutta ohjelmointikieltä voisi rauhassa opetella samalla kun aloittaisi projektin. Vanhankin projektin yhteydessä Saarenpää jatkaisi Swift-kielellä, mutta hän ei osannut sanoa tästä enempää, koska ei osaa vielä Swift-ohjelmointikieltä. Vaihtaminen Swift-ohjelmointikieleen sujuisi hänen mielestään parhaiten toteuttamalla uudet osat Swift-kielellä ja uudelleenkirjoittamisen yhteydessä käyttää myös Swift-ohjelmointikieltä. Systemaattinen uudelleenkirjoittaminen ei kuulosta houkuttelevalta, sillä edellisessäkin projektissa missä hän oli mukana, on niin paljon koodia, että työmäärä olisi vaikea perustella asiakkaalle. Hänen mielestään olisi helpompaa, jos Apple pakottaisi käyttämään Swift-ohjelmointikieltä, koska tällöin kielen käyttämisen perusteleminen asiakkaalle olisi helppoa. Tällaisen pakotteen tuleminen Applelta on kuitenkin Saarenpään mielestä hyvin epätodennäköistä. Hän uskookin, että niin kauan kuin Objective-C-kieltä voi käyttää, hänen viimeisimmässään projektissaan käytetään sitä.

5.2 Swift-projekti

Swift-ohjelmointikieli on tämän työn kirjoitushetkellä noin kolme vuotta vanha, Applen kehittämä ohjelmointikieli. Kielen tulevaisuus ei ole enää sidottu Appleen, koska kielestä on tehty avoin, ja kaikki kehittäjät voivat osallistua kielen kehittämiseen.

5.2.1 Tausta

Toinen työhön haastatelluista iOS-kehittäjistä, Koli, on käyttänyt Swift -ohjelmointikieltä reilun puolitoista vuotta samassa projektissa. Tätä projektia ennen hän on käyttänyt myös Objective-C-kieltä noin puolitoista vuotta. Koli on työskennellyt kolme vuotta iOS-käyttöjärjestelmän parissa, ja haastattelussa hän kertoo kokemuksiaan Objective-C-kielen käyttämisestä ja siirtymisestä Swift-kieleen.

Haastattelun alusta asti on selvää, että Koli on todella tyytyväinen Swift-ohjelmointikieleen eikä käytä Objective-C-kieltä kuin pakon edessä. Mikä sitten tekee Kolin mielestä Swift-ohjelmointikielestä niin paljon paremman kuin Objective-C-kieli? Swift-kielen syntaksi on paljon miellyttävämpi kuin Objective-C-kielen ja sen takia sitä on Kolin mukaan mukavampi sekä kirjoittaa, että lukea. Yksi konkreettinen asia minkä vuoksi Koli kokee Swift-ohjelmointikielen parempana kuin Objective-C-kielen on se, kuinka funktioita kutsutaan. Objective-C-kielessä käytetään funktioiden kutsumiseen kulmasulkeita, ja Swift-ohjelmointikielessä pistettä. Kulmasulkeita käytetään Swift-ohjelmontikielessä oikeastaan vain kun halutaan indeksoida säiliötä. Objective-C-kielessä funktioiden kutsuminen tapahtuu kulmasulkeiden avulla, joten ketjutetussa kutsussa voi olla lopussa ketjun pituudesta riippuen pitkä rivi sulkevia kulmasulkeita.

Kysyttäessä Swift-ohjelmointikielen tärkeimpiä ominaisuuksia, antaa Koli seuraavanlaisen listan: funktionaaliset ominaisuudet, protokollat ja optionaalisuus. Nämä kolme ominaisuutta ovat Kolin mielestä tärkeimmät. Edellä mainituista ainoastaan protokollat ovat olemassa Objective-C-kielessä, mutta ne eivät ole yhtä ilmaisuvoimaisia kuin Swift -ohjelmontikielessä. Apple on painottanut protokollia ja luonut uuden protokollaperusteisen ohjelmoinnin käsitteen. POP:in on tarkoitus olla samalla abstraktiotasolla kuin olio-ohjelmointi. Viime vuosina OOP on ollut ratkaisu, miten asioita abstrahoidaan ja mihin nimensäkin puolesta Objective-C-kieli tukeutuu vahvasti. OOP:lla on kuitenkin ollut omat ongelmansa, ja POP:in on tarkoitus tarjota vaihtoehto OOP:lle. POP:in tarkoitus ei ole kuitenkaan syrjäyttää OOP:ta, vaan olla apuna ja täydentää OOP:n periaatteita. Tämän lisäksi Swift -ohjelmontikielessä on mahdollista käyttää funktionaalista ohjelmointia. Funktionaalinen ohjelmointi on samalla abstraktiotasolla kuin imperatiivinen ohjelmointi. Tässä, kuten OOP :in ja POP:in tapauksessakin, funktionaalisen ohjelmoinnin ei ole tarkoitus syrjäyttää

imperatiivista, vaan olla sen tukena ja antaa lisää ohjelmointimahdollisuuksia.

Koli pitääkin POP -paradigmaa mielenkiintoisena ja tärkeänä. Hän uskoo, että Apple panostaa siihen tulevaisuudessa, ja pitää sitä senkin vuoksi erityisen tärkeänä. Koli kommentoi, että POP -paradigman avulla voidaan periyttäminen tehdä oikein. Tällä hän viittaa siihen, että POP:in avulla voidaan kontrolloida paremmin mitä kaikkea ”periytetään” (lainausmerkit siksi, koska tarkalleen ottaen POP ei periytä, vaan on olemassa rajapinta, jonka voi toteuttaa joko osittain tai kokonaan). On voitu määritellä, mitkä on pakko toteuttaa tai mitä ei voi enää uudelleen toteuttaa. Tämän vuoksi Koli kommentoi, että periyttäminen tehty oikein. Kolin mielestä POP:iin kannattaa panostaa ja tutustua todella hyvin, koska tulevaisuudessa POP tulee hänen mielestään olemaan todella iso asia. Nykyisinkin se on suuri osa sitä, miten Swift -ohjelmointikieltä tulee käyttää.

Koli nostaa esille vielä yhden asian Swift -ohjelmointikielestä: sen turvallisuuden. Koli viittaa siihen, että Swift -ohjelmointikielessä kääntäjä kertoo virheistä enemmän kuin Objective-C-kielessä, jossa ongelmiin törmätään vasta ajon aikana. Silloinkin virhe saattaa jäädä huomaamatta, koska funktion kutsuminen **NULL**-pointerin läpi on sallittua eikä aiheuta mitään ongelmia. Koli näkee virheiden löytämisen käännösaikana paljon parempana kuin ajon aikana. Virheet löytyvät nopeammin, ja ne ovat siitä syystä helpompia korjata. Virheiden löytäminen on helpompaa, koska kääntäjä näyttää rivin, missä ongelma on verrattuna siihen, että sovellus vain kaatusi jossain vaiheessa. On tietysti mahdollista, että debuggerin avulla voitaisiin myös selvittää millä rivillä virhe on, mutta monisäikeisen järjestelmän tapauksessa tämä ei välttämättä olisi niin yksikertaista.

5.2.2 Henkilökohtaiset mieltymykset

Seuraavassa Koli kertoo henkilökohtaisista mieltymyksistään ja vertailee Swift-ohjelmontikieltä Objective-C-kieleen. Kokemukset pohjautuvat hänen kolmen vuoden iOS-kokemukseensa, joista on puolet hän on käyttänyt Swift-ohjelmontikieltä ja puolet Objective-C-kieltä.

Syntaksi on Swift-kielessä Kolin mielestä paljon parempi verrattuna Objective-C-kieleen. Kolin mielestä syntaksi on paljon tiiviimpää verrattuna Objective-C-kieleen. Yksi suuri syy syntaksin tiiviyteen on kulmasulkeiden puuttuminen, koska Swift-ohjelmontikielessä käytetään pistenotaatiota funktioiden kutsumiseen. Koli myös luonnehtii syntaksin olevan helppoa, ja kun käyttää funktionaalisia ominaisuuksia siitä tulee lyhyempää kuin Objective-C-kielessä. Lyhyt tarkoittaa tässä yhteydessä sitä, kuinka paljon merkkejä ja rivejä tulee lähdekoodiin. Merkkien määrä vähenee, mutta myös niin sanottujen one-linereiden[1] avulla voidaan vähentää sekä merkkien

[1]Yhdellä rivillä tehdään jonkin asia missä tapahtuu monta asiaa.

määrää että rivien määrää. Koli on huolissaan siitä ovatko one-linerit haitallisia ymmärrettävyyden kannalta. Toisaalta mikäli one-linerit yleistyvät, eivätkä enää ole vain yksittäisten kehittäjien tekemiä, saattavat ne muuttua yleisesti tunnetuiksi tavoiksi hoitaa asioita. Tämän jälkeen one-linerit eivät ainakaan haittaisi ymmärrettävyyttä. Koli huomauttaa, että Swift-ohjelmontikielen one-linerit ovat eri asia kuin Objective-C-kielen one-linerit, ja näitä ei tulisi vertailla juurikaan keskenään. Swift-kielessä voidaan tuottaa one-linereita funktionaalisten ominaisuuksien avulla, ja näin ollen ne eivät ole verrattavissa imperatiivisen kielen one-linereihin.

Yksi konkreettinen esimerkki rivimäärän lyhenemisestä on if-lauseiden väheneminen. If-lauseiden käyttäminen on Kolin mukaan vähentynyt, koska saman toimenpiteen voi tapauksesta riippuen toteuttaa funktionaalisesti tai optionaalisuuteen liittyvien ominaisuuksien avulla. Funktionaaliset ominaisuudet kuten **.map** ja **.reduce** ovat siis syrjäyttäneet if-lauseen Kolin koodissa. Optionaalisia arvoja voidaan purkaa if-let-rakenteen avulla, mutta tämäkin voidaan korvata **.map**-mekanismin avulla, ja Koli kertookin käyttävänsä tätä ennemmin kuin if-let-rakennetta.

Kysyttäessä Swift-kielen eniten käytettyjä ominaisuuksia Koli nostaa esille asioita, joista on keskusteltu tässä jo aiemmin. Optionaalisuus on eniten Kolin käyttämä ominaisuus, koska se vaikuttaa kaikkiin niihin muuttujiin/vakioihin mitä hän käyttää. Toinen Kolin paljon käyttämä ominaisuus on protokollat, joista oli paljon puhetta edellisessä osiossa. Funktionaalisuus oli Kolille uusi asia kun hän alkoi käyttämään sitä Swift-ohjelmointikielen yhteydessä. Hän on hyvin innostunut funktionaalisuudesta ja aikoo panostaa funktionaaliseen ohjelmointiin tulevaisuudessa opettelemalla sitä lisää. Koli näkee funktionaalisuudesta olevan hyötyä rivimäärän lyhenemisen vuoksi, mutta samalla hän miettii tuleeko rivejä jo liian vähän eli one-linereiden määrä ja niiden ymmärrettävyys arveluttaa. Tehokkuus on yksi asioista, jotka pitää ottaa huomioon funktionaalista ohjelmointia käytettäessä. Koli kertoo, että hänen nykyisessä projektissaan eräs kohta piti muuttaa funktionaalisesta toteutuksesta imperatiiviseksi, koska kohta oli liian tehoton. Hän korostaa, että tämä on melko harvinaista, sillä hän on vain kerran puolentoista vuoden aikana joutunut tekemään niin. Tehokkuus on hyvä pitää kuitenkin mielessä kun kirjoittaa funktionaalisia toteutuksia.

Koska Koli on käyttänyt Swift-kieltä vasta puolitoista vuotta, on hänellä vielä opeteltavaa. Kielen kehittyminen ja uusien ominaisuuksien lisääminen luo tarpeen uusien asioiden opettelemiselle. Apple julkisti WWDC-tapahtumassa tänä vuonna[2] uuden, kolmannen version Swift-ohjelmointikielestä, joka tulee syksyllä kaikkien saataville. Kolmas versio tuo tullessaan lisää ominaisuuksia POP-ohjelmointiin. Koli mainitseekin haluavansa opetella lisää POP-ohjelmointia. Tällä hetkellä Kolista tuntuu, että hän on oppinut yhden tavan käyttää POP-ohjelmointia eikä hän vielä

[2] 2106

osaa soveltaa sitä oikein tai ymmärrä sen kaikkia mahdollisuuksia, koska hän katsoo asiaa toistaiseksi vain yhdeltä kantilta. Koli aikoo panostaa POP-ohjelmointiin, jotta hän osaisi tulevaisuudessa soveltaa ja käyttää sitä paremmin ja pystyisi ottamaan täyden hyödyn irti sen ominaisuuksista.

5.2.3 Huolet / Ongelmat

Ensimmäiset koodit mitä Koli kirjoitti Swift-ohjelmointikielellä olivat hänen kertomansa mukaan ”vähän jänniä”. Tällä hän tarkoittaa ettei hän heti osannut käyttää uutta ohjelmointikieltä, vaan paljon on opittu matkan varrella. Aloittaminen oli kuitenkin helppoa eikä suurempia ongelmia esiintynyt.

Oppiminen on Kolin kohdalla tapahtunut yrityksen ja erehdyksen kautta. Hän on oppinut Swift-ohjelmointikielen käyttämällä sitä. Koli kertoo lukeneensa Applen virallisen dokumentaation sekä yhden kirjan osittain, mutta ei näe niitä kovin oleelliseksi oppimisprosessissaan, koska hän ei edes enää muista lukemansa kirjan nimeä. Hänen mielestään kirjojen käyttäminen oppimiseen on ongelmallista, koska uusi kieli kehittyy niin nopeasti etteivät kirjat pysy mukana kehityksessä. Swift-ohjelmointikieli ei ole taaksepäin yhteensopiva kaikissa tapauksissa. Tämä siitä syystä, että ei haluta taakaksi alussa tehtyjä päätöksiä, jotka pitkässä juoksussa saattaisivat osoittautua ongelmallisiksi. Tulee muistaa, että kieli on ollut olemassa nyt reilut kaksi vuotta. Koli kertoo, että välillä hänellä tuntuu olevan kiire pysyä kielen kehityksen perässä ja kuinka vanhoja koodeja tulee päivittää vastaamaan uutta versiota Swift-ohjelmointikielestä. Koli näkee nuorta ohjelmointikieltä käyttäessään muutoksien tekemisen enemmän oppimisena kuin taakkana. Koli kertoo saaneensa välillä ns. ahaa-elämyksiä muuttaessaan omia vanhoja toteutuksiaan vastaamaan uutta Swift-versiota tai oppiessaan käyttämään kieltä paremmin. Kysyttäessä kaipaako Koli jotain Objective-C-kielen ominaisuutta, ei hänelle heti tullut mieleen mitään. Myöhemmin hän kuitenkin palasi asiaan ja kertoi Objective-C-kielessä olevasta key-value-observing[3]-ominaisuudesta, jonka avulla pystyy tarkkailemaan ja saamaan tiedon mikäli muuttujan arvo muuttuu. Tähän hän lisää, että asia voidaan hoitaa eri lailla Swift-ohjelmointikielessä.

iOS-alustalle kehitettäessä käytetään Xcode-kehitysympäristöä. Objective-C-kieltä pystyy ilmeisesti tuottamaan myös AppCode-nimisellä ohjelmalla, mutta Koli on aina käyttänyt Xcode-kehitysympäristöä. Kun Swift-ohjelmointikieli julkaistiin kesällä 2014, oli Xcode silloin puutteellinen. Kolin mukaan kääntämisessä esiintyi ongelmia, ja erilaisia lastentauteja oli havaittavissa. Tästä on tultu pitkä matka viimeisen kahden vuoden aikana. Lastentaudit ovat hävinneet kehitysympäristöstä, ja luottamus Xcode-kehitysympäristön käyttämiseen Swift-ohjelmointikielellä kehittä-

[3]muuttujan tarkkailusta.

miseen on hyvällä tasolla. Xcode-kehitysympäristön uuden version yhteydessä on monesti esitelty uusia ominaisuuksia, jotka saattavat sisältää pieniä ongelmia, mutta yleisesti ottaen asiat ovat hyvällä ja turvallisella mallilla. Yksi Xcoden puutteellisista toiminnallisuuksista on automaattinen täydennys, mutta tähänkin on tullut paljon parannusta alkuajoista. Koli korostaa, että mikäli asiat tehdään oikein eli ns. Swift-tyypisesti, niin ominaisuus toimii hyvin. Myös kehittäjällä on vastuuta siitä kuinka hyvin ominaisuus toimii.

Koska Swift-ohjelmointikieli on uusi, herää kysymys ovatko kolmannet osapuolet päässeet mukaan ja/tai pysyneet mukana kehityksessä. Sovelluskehityksessä iOS-alustalle käytetään projektista riippuen kolmannen osapuolen kirjastoja. Koli kertoo, että tämä ei ole enää ongelma, koska kirjastojen toteuttajat ovat joko toteuttaneet kirjastot uudestaan Swift-ohjelmointikielellä tai tarjoavat rajapintaa, jonka avulla kirjastoja voidaan käyttää Swift-ohjelmointikielen kanssa. Toinen on avun saaminen internetistä. Stackoverflow-sivusto on yksi suosituimmista sivustoista, joista etsitään apua. Sivustolta alkaa löytymään kattavasti ratkaisuja ongelmiin, joihin Koli on törmännyt Swift-ohjelmointikielen kanssa. Hän kertoo, että sivustolla on paljon laadukkaita ohjeita ja koodinäytteitä kuinka muuttaa Objective-C-kieltä Swift-ohjelmointikieleen. Kaikkein laadukkaimmissa on esitelty ja näytetty kuinka asia tulee Swift-ohjelmointikielellä toteuttaa eikä vain miten syntaksi muutetaan Objective-C-kielestä Swift-ohjelmointikieleen.

5.2.4 Projekti

Teknisiä rajoitteita Swift-ohjelmoitikielen käyttämiseen on vain iOS-käyttöjärjestelmän versio. Mikäli halutaan käyttää iOS 7:ää, tulee kolmannen osapuolen kirjastot kopioida projektiin ja Swift-kirjastojen käyttäminen ei ole mahdollista.

Kehittäjän kannalta Swift-ohjelmointikielen käyttämistä saattaa Kolin mukaan rajoittaa tilanne, missä henkilö, joka ei ole käyttänyt ollenkaan Swift-ohjelmointikieltä joutuu käyttämään sitä tiukassa aikataulussa. Tässäkin tapauksessa kehittäjän kyky oppia uutta näyttelee merkittävämpää roolia kuin Swift-ohjemointikieli. Kolin mielestä Swift-kieli on helppo opetella, varsinkin jos osaa Objective-C-kielen ja iOS-alusta tuttu - iOS-osaaminen on yhteistä, käytetään sitten kumpaa tahansa ohjelmointikieltä.

Kolin mielestä Swift-ohjelmointikielen käyttäminen on nopeampaa kuin Objective-C-kielen. Tämä johtuu one-linereista, syntaksin helppoudesta sekä siitä, kuinka tiivistä Swift-ohjelmointikieli on Objective-C-kieleen verrattuna. Swift-ohjelmointikieltä käytettäessä tulee vähemmän merkkejä ja rivejä ja se on myös ilmaisuvoimaisempi, mikä tekee siitä nopeampaa käyttää. Koli on jutellut muiden projektissa olleiden kanssa, jotka myös kokevat, että projekti on edennyt nopeammin, koska he ovat käyttäneet Swift-ohjelmointikieltä. Huomion arvoista on, että projekti oli en-

simmäinen, missä Koli on käyttänyt Swift-ohjelmointikieltä. Hän on opetellut käyttämään Swift-ohjelmointikieltä vasta projektin aikana, ja silti sen käyttäminen on ollut nopeampaa kuin tutun Objective-C-kielen käyttäminen.

5.2.5 Tulevaisuus

Haastattelun aikana on kävi selväksi, että Kolin mielestä Swift-ohjelmointikieli on tulevaisuus ja siihen tulee panostaa. Objective-C-kieli on vielä mukana jossain määrin, mutta sitä ei kannata enää alkaa opettelemaan. Niidenkin kehittäjien, jotka eivät ole vielä tutustuneet Swift-ohjelmointikieleen on korkea aika aloittaa siihen tutustuminen. Swift-ohjelmointikieli kehittyy todella nopeasti, koska sen kehittäminen on hyvin aktiivista. Swift-ohjelmointikielen tulevaisuus on taattu nyt kun kielestä on tehty avoin ja jokainen, joka haluaa voi osallistua sen kehittämiseen. Tällä hetkellä yhteisö on todella aktiivinen, vaikka Swift-ohjelmointikieli on vasta vähän aikaa sitten muutettu avoimeksi. Tulevaisuus vaikuttaa siis lupaavalta.

Avoimuuden myötä Koli uskoo Swift-ohjelmointikielen leviävän tulevaisuudessa aluksi palvelinkoodiin sekä jossain vaiheessa myös mahdollisesti kaikille alustoille. Hän näkee, että Swift-ohjelmointikielessä on potentiaalia tulla yhdeksi suosituimmaksi ja käytetyimmäksi ohjelmointikieleksi. Nykyään kehitys on rajoittunut oikeastaan iOS- ja macOS-alustoille. Koli kertoo kollegastaan, joka oli toteuttanut yksinkertaisen REST-rajapinnan käyttäen Swift-ohjemointikieltä. Aika näyttää, kuinka suosittu ja kuinka nopeasti Swift-ohjelmointikielestä tulee myös palvelinpuolella. Kolilla on toiveita kielen suhteen lähinnän liittyen Cocoa Touch-kirjastoon, joka on toteutettu Objective-C-kielellä. IOS-kehityksessä käytetään paljon Cocoa Touch-kirjastoa, sillä se on Applen tarjoama kirjasto sovelluksien kehittämistä varten. Cocoa Touch-kirjastoa käytettäessä voi helposti huomata sen olevan toteutettu Objective-C-kielellä. Tämän seurauksena lähdekoodi ei näytä Swift-ohjelmointikielimäiseltä näiltä osin. Tähän on tulossa muutos Swift-ohjelmointikielen kolmannessa versiossa, jossa Apple on toteuttanut Swift-ohjelmointikielimäiset rajapinnat Cocoa Touch-kirjastolle. Lisäksi joitain yleisesti käytettyjä ominaisuuksia kuten säikeen aloittaminen on muutettu syntaksiltaan näyttämään samalta kuin muiden funktioiden kutsuminen Swift-ohjelmointikielessä.

Yhteenvetona voidaan todeta, että Koli ei halua käyttää Objective-C-kieltä ellei ole pakko ja on todella tyytyväinen Swift-ohjelmointikieleen sekä odottaa innolla mitä ohjelmointikielen tulevaisuus tuo tullessaan.

5.3 Swift- ja Objective-C-yhdistelmäprojekti

Kolmas työhön haastateltu iOs-kehittäjä, Määttä, on käyttänyt Swift-ohjelmointikieltä nyt kaksi vuotta yhdistelmäprojektissa. Projektissa on sekä Objective-C-kieltä

ja Swift-ohjelmointikieltä. Objective-C-kielestä Määtällä on noin neljän kuukauden kokemus aikaisemmasta projektista ennen nykyistä projektia. Nykyisestä projektista päätettiin tehdä yhdistelmäprojekti lähinnä tiimin osaamisen vuoksi. Kehittäjä, joka teki projektia aluksi yksin, oli ehtinyt aloittaa projektin Objective-C-kielellä ennen kuin siihen liittyi mukaan toinen henkilö. Toisen henkilön myötä projektista tuli yhdistelmäprojekti, koska mukaan tullut kehittäjä osasi Swift-kieltä. Nykyinen tilanne on, että molemmat alkuperäiset kehittäjät ovat siirtyneet muihin projekteihin ja Määttä on jäänyt jatkokehittämään / ylläpitämään projektia. Yhdistelmäprojekti, jossa on vain yksi Swift-osaaja aiheuttaa tilanteen, jossa ei ole pätevää henkilöä katselmoimaan Swift-koodeja. Tämä on ongelma, koska koodikatselmoinnilla on positiivinen vaikutus lähdekoodin laatuun.

5.3.1 Swift-kokemukset

Määtällä on noin kahden vuoden kokemus Swift-ohjelmointikielestä. Hän kuvailee sen olevan lähempänä C-perhettä ja Java-kieltä kuin Objective-C-kieltä, ja helpompi hänelle koska hän omaa vahvan C-perhetaustan. Swift-kieleen on hänen mielestään helpompi palata pitkän ajan kuluttua kuin Objective-C-kieleen, koska se muistuttaa C-perheen kieliä. Tämä on ehkä yleinenkin mielipide, koska C-perhe ja Java-kieli ovat käytetyimpiä ohjelmointikieliä. Swift-kielestä on myös tehty paljon modernimpi verrattuna Objective-C-kieleen. Tämä on ymmärrettävää, koska Objective-C-kieli on kehitetty noin 30 vuotta sitten, ja Swift on kaksi vuotta vanha, mutta Määttä vertaa tässä Swift-ohjelmointikieltä kaikkiin nykyään olemassa oleviin ohjelmointikieliin.

Geneerisyys on ominaisuus, josta Määttä on erityisesti pitänyt Swift-ohjelmointikielessä verrattuna Objective-C-kieleen. Kun geneerisyyteen vielä yhdistetään protokollat ja laajennokset, on Swift-kieli Määtän mielestä ylivertainen verrrattuna Objective-C-kielen mekanismeihin tuottaa geneeristä koodia. Toisaalta hän on myös huomannut kääntöpuolena, että käännösajat ovat kasvaneet Swiftiä käytettäessä. Määttä epäilee tämän johtuvan tietotyyppien päättelystä, jota kääntäjä joutuu tekemään geneerisen koodin tapauksessa hyvin paljon. Hän on huomannut, että käännösajat lyhentyvät mikäli hän auttaa kääntäjää tilanteissa missä tietotyyppi tiedetään. Reflektiivisyys on asia mikä puuttuu Swift-kielestä, ja jota Määttä siihen kaipaisi. Reflektiivisyys on Määtälle tuttua Java-kielestä. Sen avulla olisi Määtän mukaan helppo parsia JSON-objekteja kielen omiin tietotyyppeihin vaivattomasti. Java-kielessä on myös toinen ominaisuus, mistä Määttä pitää: roskienkeruu. Roskienkeruun idea on, että muistinhallinta käy kaikki oliot läpi ja tarkistaa viittaako niihin mikään ja jos ei viittaa, niin kieli poistaa kyseisen olion muistista. Tämä on hyvin kiistelty muistinhallintamekanismi. Puolustajat vetoavat edellä mainittuun ominaisuuteen, kun taas vastustajat karsastavat roskienkeruuta, koska se saattaa ai-

heuttaa yllättäviä kuormia hetkellisesti. Näitä kuormituksia ei voida ennustaa eli ei tiedetä milloin roskienkeruu tapahtuu. Swift-ohjelmontikielessä käytetään viitteiden laskemista. Idea on, että mikäli muistiin ei osoita kukaan, voidaan se poistaa. Tämä on kuitenkin ongelmallista. Viittaamisia on kahden tyyppisiä: vahvoja ja heikkoja.

Objective-C-kielessä on lohkot, ja Swift-ohjelmointikielessä on sulkeumat. Määttä pitää syntaktisesti enemmän Swift-ohjelmointikielen sulkeumista ja erityisesti siitä, että sulkeuman ollessa viimeinen argumentti voidaan se emittoida3.4 . Swift-kielen yhteydessä on puhuttu paljon sen funktionaalisista3.3 ominaisuuksista, mutta Määttä ei ole niistä kovin innostunut, koska ne ovat hänelle tuttuja hänen aikaisemmin käyttämistään kielistä. Innostuminen funktionaalisesta ohjelmoinnista on siis tapahtunut jo toisen ohjelmointikielen yhteydessä. Hän kokee funktionaalisten ominaisuuksien olevan tärkeitä ja hyödyllisiä. Objective-C-kielessä funktionaalinen ohjelmointi ei ole mahdollista kielen tasolla kuten Swift-ohjelmointikielessä. Osalla säiliöistä on omia vastaavia ominaisuuksia.

Koska Swift-kieli ja samalla sen kääntäjä ovat uusia, on Määttä huomannut ongelmia kääntäjän käyttämisessä. Edellä on jo mainittu, että käännösajat saattavat pitkittyä, jos kirjoittaa paljon geneeristä koodia. Toinen huomio on, että puhtaan käännöksen ajaminen[4] saattaa poistaa joitain käännösvirheitä ja estää sovelluksen kaatumisen. Määttä huomauttaa, että nämä tilanteet ovat harvinaisia, ja ovat tulleet hänelle vastaan kattavan uudelleenkirjoittamisen yhteydessä. Asiat ovat parantuneet hänen mielestään paljon matkan varrella. Määtän suositus on puhtaan käännöksen tekeminen isomman uudelleenkirjoittamisen jälkeen tai kun on tekemässä uutta versiota jakeluun.

5.3.2 Vertailu muihin ohjelmointikieliin

Määtän mielestä Swift-ohjelmointikieli muistuttaa C-perheen kieliä ja Java-kieltä enemmän kuin Objective-C-kieltä. Suuri syy tähän on syntaksi. Hän vertaa Objective-C-kielen []-syntaksia Lisp-ohjelmointikieleen, joka on tunnettu funktionaalinen ohjelmointikieli, jossa on todella paljon ()-sulkeita verrattuna melkein mihin tahansa nykyään paljon käytettyyn ohjelmointikieleen. Objective-C-kielen opetteleminen oli Määtälle suorastaan tuskallista juuri syntaksin vuoksi. Syntaksin vuoksi myös palaaminen Objective-C-kielen pariin on paljon vaikeampaa, koska se on niin erilainen muihin Määtän käyttämiin kieliin verrattuna. Swift-ohjelmointikielen opetteleminen taas on ollut helppoa ja miellyttävää Määtälle. Syntaksi on vaikuttanut paljon kielen oppimiseen, mutta myös muut kielen ominaisuudet, kuten nil-osoittimien käsittely ovat auttaneet. Swift-kielessä **nil**-arvossa olevan muuttujan käsitteleminen aiheuttaa sovelluksen kaatumisen, mikäli sen arvo ei saisi olla **nil**. Objective-C-kielessä

[4]Suora suomennos sanoista clean build.

nil-arvoisen osoittimen läpi funktion kutsuminen on sallittua, ja se on käytäntönä Cocoa Touch-kirjastossa. Tämä on tietysti helpompaa, kun ei tarvitse välittää onko jokin arvossa **nil** vai ei, mutta sillä on sivuvaikutuksia. Sovellus ei kaadu ajon aikana tähän Objective-C-kielen tapauksessa, jonka seurauksena ja mikäli tämä on virhe ei sitä saada kiinni vaan sovellus jatkaa suorittamista johonkin asti ja johonkin on ennalta määrittelemätön. Määttä korostaa tämän olevan hyvin epäintuitiivista C-perhetaustaiselle. Swift-ohjelmointikielessä on optionaaliset3.2.5, joita Määttä pitää erityisen hyvinä, koska niiden avulla ei tule vahingossa kutsuttua **nil**-arvoa ja voidaan asettaa muuttuja ”ei mikään”-arvoon ja tarkistaa se. Swift-kielessä on rakenteita kuten **if let** ja **guard**, joiden avulla optionaalisten käsittely on helppoa ja vaivatonta. On hyvä huomata, että optionaalisuuden käyttäminen ei ole pakollista eli kehittäjä voi tehdä itse Swift-ohjelmointikielessä ratkaisut tämän suhteen.

Poikkeuksien käsittely on Swift-ohjelmointikielessä paljon paremmalla tasolla kuin Objective-C-kielessä, Määttä kertoo. Objective-C-kielessä on käytössä **NSError**, joka on hänen mielestään ”suorastaan järkyttävä” käyttää. Tieto pitää lisätä **userinfoon**, joka on map-tyyppinen tietorakenne, jossa on avaimena itse keksitty merkkijono. Hän on todennut, että koska ne ovat niin huonoja käyttää, hän ei käytä niitä ollenkaan. Swift-ohjelmointikielen poikkeuksia hän taas kertoo käyttävänsä, koska ne ovat helppoja. Niihin voidaan sijoittaa tietueiden avulla tietoa, ja laajennuksien avulla mahdollisuudet ovat rajattomat. Funktionaalinen ohjelmointi on tuonut paljon niin sanottuja ”one-liner´ -koodirivejä Swift-lähdekoodiin eli jokin asia on hoidettu yhdessä rivillä. Koodirivejä tulee näin vähemmän, mutta se ei kuitenkaan takaa ylläpidettävyyttä. Toisaalta koska funktionaaliset ”one-liner” -koodirivit ovat kaikille tuttuja, ovat ne Määtän mukaan helpompia ymmärtää kuin Objective-C-kielellä tehdyt ”one-liner” -rivit. Määttä myös kirjoittaa for-toistorakenteen sijaan funktionaalisen toteutuksen, joka on todennäköisesti ”one-liner”, koska siitä näkee hänen mielestään paremmin mitä tapahtuu. Tämä siksi, ettei toistorakenteen yksityiskohtia tarvitse miettiä, vaan voi keskittyä itse toistorakenteen sisällä olevaan asiaan.

5.3.3 Kielen valinta projektin kannalta

Uudessa projektissa Määttä käyttäisi Swift -ohjelmointikieltä ennemmin kuin Objective-C-kieltä. Hän karsastaa ajatusta Objective-C-kielen käyttämisestä. Swift-ohjelmointikieli on tutumpi hänelle, ja hän on käyttänyt sitä enemmän kuin Objective-C:tä, mutta Swiftin valintaan on muitakin syitä. Seuraavaksi käydään läpi, miten Määtän mielestä kannattaa lähestyä kielen valintaa uuden projektin alkaessa.

Asiaa kannattaa ennen kaikkea lähestyä kehittäjien kannalta. Mikäli talossa on paljon Objective-C-kielen osaajia, jotka eivät ole innokkaita opettelemaan uutta ohjelmointikieltä (tässä tapauksessa Swift -ohjelmointikieltä), kannattaa valita Objective-

C-kieli. Määttä kuitenkin huomauttaa, että Swift-kielen opetteleminen olisi Objective-C-kielen osaajille nopea ja helppo prosessi iOS-kehittämisen kannalta, koska kummassakin kielessä käytetään samaa Cocoa Touch-kirjastoa ja muutkin iOS-kehittämisen paradigmat ovat kielissä samat. Määtällä kului päivä Swift -ohjelmointikielen oppimiseen. Hän luki Applen tekemän tiivistelmän Swift-ohjelmointikielestä, jonka jälkeen hän alkoi seuraavana päivänä käyttää sitä projektissa. Tässäkin yhteydessä Määttä korostaa C-perhetaustaansa, joka hänellä oli helpottamassa oppimista. Syntaksi on Swift-ohjelmointikielessä hänen mielestään yksikertainen ja helppo oppia, minkä vuoksi koko kielen oppiminen on vaivatonta.

Apple on selvästi panostanut paljon uuteen ohjelmointikieleensä, ja siksi vanhan Objective-C-kielen tulevaisuus on määrittelemätön. Apple ei ole kertonut, mitä Objective-C-kielelle tapahtuu tulevaisuudessa. Määttä ei kuitenkaan usko, että tästä pitäisi olla huolissaan. iOS-sovelluksien elinkaari ei ole hänen mielestään niin pitkä, että Apple tiputtaisi tässä ajassa Objective-C-kielen kokonaan pois. Määttä suosittelee Swift-kielen käyttämistä Objective-C:n sijasta, mikäli projektissa on aikaa, ja kehittäjät ovat siihen halukkaita. Kehittäjät ovat kielen valinnassa avainasemasssa, koska sillä on merkitystä, kuinka halukkaita he ovat ja kuinka nopeasti he pystyvät oppimaan. Mikäli projektissa päätetään opetella Swift-ohjelmointikieli, on Määtän mielestä tärkeää, että kaikki koodit katselmoidaan ja yksi kehittäjistä osaa jo käyttää hyvin Swift-kieltä. Muuten on vaarana, että projektissa kirjoitetaan Swift-ohjelmointikielen syntaksilla Objective-C-lähdekoodia. Hänen kohdallaan Swift-kielen oppiminen tapahtui muiden esimerkin avulla ja saamalla palautetta omasta lähdekoodistaan. Se on ollut Määtälle paljon parempi tapa oppia kuin yrittää itse soveltaa jotain lukemaansa.

On olemassa yksi tilanne, joka pakottaa käyttämään Objective-C-kieltä Swift-ohjelmointikielen sijaan: projekti, jossa tulee kutsua C++-koodia. Tämä on mahdollista Swift-kielessä vain erilaisten käärintäluokkien[5] avulla, kun taas Objective-C-kielessä voidaan kutsua suoraan C++-kieltä. Objective-C-tiedostoon voidaan jopa kirjoittaa C++-koodia. Tämä on ainut tapaus, missä Määttä on huomannut ettei Swift-ohjelmointikieltä kannata käyttää. Riippuen siitä, kuinka paljon C++-koodia tulee kutsua, kannattaa harkita kuinka paljon aikaa / vaivaa käärimiaen käyttämiseen menee.

5.3.4 Yhdistelmäprojekti

Apple on tehnyt mahdolliseksi käyttää sekä Objective-C-kieltä ja Swift-ohjelmointikieltä samassa projektissa. Teknisesti tämä ei ole vaikeaa. Objective-C-luokat pitää listata yhteen tiedostoon, joka toimii siltana ohjelmointikielien välillä. Apple viittaa

[5]Suoran suomennos sanasta wrapper.

tiedostoon ”birdging header”-nimellä. Objective-C-luokkien pitää lisätä projektin nimestä koostuva esittelytiedosto, mikäli käytetään Swift-luokkaa. Onko tästä sitten mitään hyötyä tilanteessa, jossa käytetään vain toista ohjelmointikieltä? Määtän vastaus tähän kysymykseen on ei.

Koska yhdistelmäprojekti ei tuo mitään lisäarvoa verrattuna yhdellä ohjelmointikielellä toteutettuun projektiin, ei Määttä suosittele yhdistelmäprojekteja. Hän uskoo, että yhdistelmäprojekteihin ennemminkin joudutaan kuin että ne tietoisesti valittaisiin. Ainut poikkeus tähän on edellä mainittu C++-kielen käyttäminen. Määttä on omassa yhdistelmäprojektissaan tehnyt Objective-C-kielellä toteutetuista osista kirjaston, jotta pääsisi eroon yhdistelmäprojektista. Sinänsä yhdistelmäprojekti ei tuo itse mitään ongelmia, mutta se aiheuttaa kehittäjälle paljon henkistä kuormaa. Tästä hyvänä esimerkkinä on se, että kun kehittäjä vaihtaa Objective-C- ja Swift-tiedoston välillä, syktaksi vaihtuu ja kehittäjä joutuu koko ajan miettimään, miten syntaksi meni missäkin kielessä. Tämä lisää turhaa kuormaa ja hidastaa kehitystyötä. Yhdistelmäprojektissa, jossa kielet pysyvät erillään toisistaan, ei ole tätä ongelmaa ja näin ollen yhdistelmäprojektista ei ole haittaa. Määttä suosittelee, että uudelleenkirjoittamisen yhteydessä asiat toteutettaisiin Swift-ohjelmointikielellä, jotta projekti voisi pikkuhiljaa muuttua valtaosin Swift-projektiksi. Määttä ei suosittele systemaattista uudelleenkirjoittamista, ellei ole tarkoitus opetella Swift-ohjelmointikieltä, ja projektissa ei ole mitään muuta tärkeämpää tekemistä. Tosin tässä kannattaa ottaa huomioon projektin koko, sillä pienempien projektien kohdalla uudelleenkirjoittaminen on järkevämpää kuin isojen. Mikäli uudelleenkirjoittaminen ei vaikuttaisi merkittävästi projektin kokonaiskustannuksiin, voisi systemaattista uudelleenkirjoittamista harkita. Kannattaa myös ottaa huomioon kuinka kauan projekti tulee jatkumaan. Mikäli sovelluksen käyttäminen tai projekti on ohi seuraavan viiden vuoden sisällä, ei uudelleenkirjoittamista kannata tehdä. Yksi tekijä on myös kehittäjät. Onko tulevaisuudessa käytössä Objective-C-kielen osaajia? Mikäli ei, voi olla hyvä idea kirjoittaa koko projekti uusiksi Swift-kielellä vielä niin kauan kun joku osaa Objective-C-kieltä. Uudelleenkirjoittamisen puolesta voidaan argumentoida seuraavien perusteluiden avulla: se on turvallisempi (if let, guard) ja kieli itsessään ohjaa turvallisemman koodin kirjoittamiseen (optionaalisuus). Määtän mielestä tässäkin asiassa lopullisen ratkaisun tekevät kehittäjät, kuten he tekivät uuden projektin kielen valinnan kohdallakin. Asiasta tulisi keskustella avoimesti, ja kartoittaa mitä projektissa mukana olevat kehittäjät ovat siitä ja tulevaisuudesta mieltä.

5.3.5 Työkalut

iOS-kehittämisessä käytetään Xcode-kehitysympäristöä. Tämä on yhteistä kummallekin kielelle, mutta sen toimivuudessa on jonkin verran eroja eri kielien välillä. Re-

faktorointi ei ole tällä hetkellä mahdollista Swift-lähdekoodille. SourceKit on komponentti, joka jäsentää ja auttaa korostamaan lähdekoodia eri väreillä. Joissain tilanteissa SourceKit saattaa hajota, jolloin värit häviävät. Automaattinen täydentäminen käyttää samaa komponenttia ja hajoaa samalla kun korostaminen hajoaa. Tätä tapahtuu vain Swift-ohjelmointikielen yhteydessä. Lisäksi Swift-kieltä käytettäessä Xcode-kehitysympäristö näyttää omaavan jonkin verran lastentauteja, mutta ne ovat vähentyneet Määtän mielestä hyvin paljon ajan kuluessa ja ovat nykyään harvinaisia. Tässäkin suhteessa on menty koko ajan parempaan suuntaan, ja edellä mainitut ongelmat eivät ole kielen valinnan kannalta kriittisiä, vaikkakin ne on hyvä ottaa huomioon.

6. YHTEENVETO

iOS-kehittäminen on tähän asti toteutettu käyttämällä Objective-C-ohjelmointikieltä. Objective-C-kieli on jo yli kolmekymmentä vuotta vanha[35]. Tämä ei tietenkään automaattisesti tarkoita, että ohjelmointikieli olisi huono. Käytetyimmät kielet kuten C, C++ ja Javat ovat kaikki iäkkäitä ohjelmointikieliä. Monelle Objective-C-kielen käyttäjälle kielen syntaksi on ollut luotaantyöntävä. Haastatteluissa kävi selvästi ilmi kuinka kulmasulkeilla funktioiden kutsuminen on huono syntaksi. Tämä ei ole ainut syntaktinen asia, jonka käyttäminen on ollut hankalaa kehittäjien mielestä vuosien varrella.

Swift-ohjelmointikieli on nyt[1] kolme vuotta vanha. Nykyinen versio on kolmas, koska joka kesä vuodesta 2014 alkaen Apple on julkistanut uuden version ja kasvattanut numeroa yhdellä. Kieli ei ole yhteensopiva taaksepäin, ja tämä on mielestäni hyvä. Itse yhteensopivuuden taaksepäin rikkominen ei ole niinkään hyvä asia, vaan se, että kieltä kehitetään vauhdilla ja alussa tehdyt päätökset eivät ole taakkana kehitykselle. Kieli on uusi, ja mikäli siitä halutaan tehdä yksi parhaista ja eniten käytetyistä kielistä, tulee sitä kehittää ja tehdä isoja päätöksiä, vaikka se rikkoisi nyt alussa yhteensopivuuden taaksepäin. Kieli on todella moderni, ja siinä on paljon ominaisuuksia, jotka kehittäjät ovat ottaneet ilolla vastaan ja joita he käyttävät iOS-sovelluskehityksessä. Protokollapohjainen ohjelmointiparadigma on saanut paljon suosiota kehittäjien keskuudessa, ja funktionaaliset ominaisuudet ovat otettu vastaan riemulla. Vahva tyypitys on tuonut turvallisuutta ja helpottanut ohjelmointia kun ei tarvitse erikseen esitellä tietotyyppejä mikäli kääntäjä osaa ne päätellä. Kielessä on paljon uusia, hienoja ominaisuuksia ja joitain vanhoja ominaisuuksia on kehitetty eteenpäin.

Haastatteluissa kävi ilmi, että missään projektissa ei enää kehitettäisi Objective-C-kielelä, ellei olisi pakko. Tähän on muutamia poikkeuksia mitä tulee ovat lähinnä budjettin ja ajan riittävyyteen. Swift-ohjelmointkieleen vaihtaminen saattaa tuottaa lisää työtä ja näin ollen budjetti ja aikataulu ei välttämättä kestä tätä muutosta. Objective-C-kieli on kehittäjien keskuudessa muinainen jäänne, josta halutaan mahdollisimman pian eroon. Havaintojen perusteella olenkin sitä mieltä, että mitä nopeammin voidaan siirtyä käyttämään Swift-ohjelmointikieltä iOS-sovelluskehittämisessä sen parempi. Objective-C-kieli on historiaa.

[1] 11.2016

LÄHTEET

[1] Apple. About the ios technologies. web. https://developer.apple.com/library/content/documentation/Miscellaneous/Conceptual/iPhoneOSTechOverview/Introduction/Introduction.html, viitattu 15.11.2016.

[2] Computerworld (Australia). The a to z of programming languages: Objective-c. web. http://www.computerworld.com.au/article/350272/z_programming_languages_objective-c/?, viitattu 9.2.2012.

[3] John Backus. Can programming be liberated from the von neumann style? a functional style and its algebra of programs. http://worrydream.com/refs/Backus-CanProgrammingBeLiberated.pdf, viitattu 18.10.2016.

[4] Joshua Ballanco. Learning swift programming: Is it ready for prime time? https://www.toptal.com/swift/swift-is-it-ready-for-prime-time, viitattu 16.05.2016.

[5] Ryan Block. Live from macworld 2007: Steve jobs keynote. https://www.engadget.com/2007/01/09/live-from-macworld-2007-steve-jobs-keynote/, viitattu 09.11.2016.

[6] Eric Cerney. Swift guard statement. http://ericcerney.com/swift-guard-statement/, viitattu 13.09.2015.

[7] Ward Cunningham. Hello world in many programming languages. http://c2.com/cgi/wiki?HelloWorldInManyProgrammingLanguages/, viitattu 16.05.2016.

[8] Maxime Defauw. Arc and memory management in swift. web. https://www.raywenderlich.com/134411/arc-memory-management-swift, viitattu 16.11.2016.

[9] Lucy Hattersley. Complete guide to swift 3.0's new features and announcements: What to expect at wwdc 2016. http://www.macworld.co.uk/feature/iosapps/what-apple-will-announce-with-swift-30-at-wwdc-2016-3638662/, viitattu 16.05.2016.

[10] Joe Howard. Introduction to functional programming in swift. https://www.raywenderlich.com/114456/introduction-functional-programming-swift, viitattu 18.10.2016.

[11] John Hughes. Why functional programming matters. https://www.cs.kent.ac.uk/people/staff/dat/miranda/whyfp90.pdf, viitattu 18.10.2016.

[12] Apple Inc. Apple - open source. http://www.apple.com/opensource/, viitattu 09.11.2016.

[13] Apple Inc. Apple age rating. http://techcrunch.com/2009/06/29/heres-how-iphone-app-store-ratings-work-hint-they-dont/, viitattu 10.02.2015.

[14] Apple Inc. Apple fundation documentation. web. https://developer.apple.com/reference/foundation, viitattu 08.02.21012.

[15] Apple Inc. Apple swift documentation. https://developer.apple.com/swift/, viitattu 08.02.2015.

[16] Apple Inc. Apple xcode documentation. https://developer.apple.com/xcode/features/, viitattu 08.02.2015.

[17] Apple Inc. Automatic reference counting. web. https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/AutomaticReferenceCounting.html, viitattu 14.11.2016.

[18] Apple Inc. Cocoa (touch). web. https://developer.apple.com/library/content/documentation/General/Conceptual/DevPedia-CocoaCore/Cocoa.html, viitattu 14.11.2016.

[19] Apple Inc. Core os layer. web. https://developer.apple.com/library/content/documentation/Miscellaneous/Conceptual/iPhoneOSTechOverview/CoreOSLayer/CoreOSLayer.html, viitattu 15.11.2016.

[20] Apple Inc. Core services layer. web. https://developer.apple.com/library/content/documentation/Miscellaneous/Conceptual/iPhoneOSTechOverview/CoreServicesLayer/CoreServicesLayer.html, viitattu 15.11.2016.

[21] Apple Inc. ios - ios 10 - apple. http://www.apple.com/ios, viitattu 09.11.2016.

[22] Apple Inc. ios developer enterprise program. https://developer.apple.com/programs/ios/enterprise, viitattu 08.02.2015.

[23] Apple Inc. ios developer license. web. https://developer.apple.com/support/ios/program-renewals.php, viitattu 08.02.2015.

[24] Apple Inc. ios developer program. web. https://developer.apple.com/programs/ios/, viitattu 08.02.2015.

[25] Apple Inc. ios developer university program. https://developer.apple.com/programs/ios/university/, viitattu 08.02.2015.

[26] Apple Inc. Llvm compiler overview. https://developer.apple.com/library/content/documentation/CompilerTools/Conceptual/LLVMCompilerOverview/, viitattu 09.11.2016.

[27] Apple Inc. Media layer. web. https://developer.apple.com/library/content/documentation/Miscellaneous/Conceptual/iPhoneOSTechOverview/MediaLayer/MediaLayer.html, viitattu 15.11.2016.

[28] Apple Inc. Swift and objective-c in the same project. https://developer.apple.com/library/ios/documentation/Swift/Conceptual/BuildingCocoaApps/MixandMatch.html, viitattu 16.05.2016.

[29] Apple Inc. Swift has reached 1.0. https://developer.apple.com/swift/blog/?id=14, viitattu 09.11.2016.

[30] Apple Inc. tvos - apple. http://www.apple.com/tvos/, viitattu 09.11.2016.

[31] Apple Inc. watchos - apple. http://www.apple.com/watchos/, viitattu 09.11.2016.

[32] Sharp Five Software Inc. Should your team switch to swift. https://sharpfivesoftware.com/2015/04/21/should-your-team-switch-to-swift/, viitattu 16.05.2016.

[33] Stas Kirichok. Is swift faster than objective-c? https://yalantis.com/blog/is-swift-faster-than-objective-c/, viitattu 16.05.2016.

[34] Danijel Lombarović. Functional programming in swift. http://five.agency/functional-programming-in-swift/, viitattu 16.05.2016.

[35] Matti Riihimäki. *Ohjelmointikielt OBJECTIVE-C ja C++ erot.* 2012.

[36] Jordan Schaenzle. Working with objective-c and swift on the same project. https://spin.atomicobject.com/2015/09/24/swift-to-objective-c/, viiattu 16.05.2016.

[37] Anthony Schmieder. Swift, c++ performance. http://www.primatelabs.com/blog/2014/12/swift-performance/, viitattu 16.05.2016.

[38] Paul Solt. Swift vs. objective-c: 10 reasons the future favors swift. http://www.infoworld.com/article/2920333/mobile-development/swift-vs-objective-c-10-reasons-the-future-favors-swift.html, viitattu 16.05.2016.

[39] Mattt Thompson. Nshipster swift operators. http://nshipster.com/swift-operators/, viitattu 26.08.2015.

[40] TIOBE. Tiobe index for may 2016. http://www.tiobe.com/tiobe_index, viitattu 16.05.2016.

[41] Éric Lévénez. Computer languages history. web. http://www.levenez.com/lang/, viitattu 9.2.2012.